University of Canterbury

Development and analysis of computer vision solution concepts using game engine based simulations in visually realistic environments.

A thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy

Ori Ganoni

supervised by
Associate Professor Dr R. Mukundan
Professor Dr R Green
Associate Professor Dr Andreas Willig

August 25, 2019

Abstract

Robotic platforms such as drones, (ROVs), and semi-autonomous vehicles are becoming more prevalent. The need for testing and analyzing techniques for algorithms and experimentation increases accordingly. For the goal of fully autonomous vehicles data gathering, annotating is a major obstacle. It requires resources for collecting data such as vehicles, cameras and sensors in addition to the manual effort of annotating the data for some of the applications. For that reason, most of the publicly available datasets and research work is carried out in common domains such as urban city space. A possible solution to this problem can be using visually realistic simulation. In this research, we will analyze the contribution of game engines as a close to realistic simulation medium. Our purposed methods involve using an existing game engine with existing assets originally developed for gaming to create a simulated visual environment for the purpose of conducting scientific experiments taking advantage of the visualisation power of the game engine. The current state of the art and latest technologies in game engines and computer graphics will also be considered. Using our approach will give researchers in the computer vision domain a valuable tool for testing their algorithms in high fidelity manner. New research tools will be presented such as frameworks for conducting experiments inside a game engine. Our tools will also provide capabilities for manipulating the simulated scene such as variable wind strength and lighting conditions. Two previously highly challenging unexplored simulation domains will be used as the main environments for conducting our experiments. One will be the drone simulation domain in a natural environment and the second will be the underwater ROV simulation domain. Both domains will be thoroughly explored. New visual simulation models and assets, as well as dynamic vehicle models that were developed for this research, will be presented as part of the simulated environments and frameworks. Existing computer vision algorithms and new algorithms developed for the purpose of this research will be tested in our simulation and in a real-world environment for comparison. The simulated experiments will be "end to end" experiments including fully simulated hardware, physics, inertial sensors and cameras. The experiments done in the simulation will be used to develop and fine-tune new methods for computer vision based navigation in those challenging domains. By showing the similar performance of a successful vehicle manoeuvre in simulation compared to a real environment using a newly developed computer vision algorithm we will validate the usefulness of game engines based simulation for computer vision research. The results show that our approach provides computer vision and autonomous vehicle researchers a new valuable tool for testing and analyzing algorithms by utilizing the visualization power of state of the art game engines in new highly challenging environments.

Acknowledgements

- I want to thank my supervisor Professor Mukundan for his close supervision, support and guidance throughout the period of the thesis. Regular weekly meeting and generally open door were important to me as a new researcher. His guidance in our joined publications was priceless.
- I want to thank my co-supervisor Professor Green for his close supervision, open door approach, support in funding and for promoting my ideas throughout my entire research period. In addition to funding, he provided me with additional team support for complex multidisciplinary tasks required by my research.
- I want to thank Tim Ransen who worked closely with me on the mechanical aspects of my research. His design has helped us achieve some of our primary goals of this research.
- I want to thank Chris Cornelisen from the Cawthron Institute for help funding our research as part of the Callaghan Innovation grant, CRS-S7-2017 Precision Farming Technology for Aquaculture.
- I want to thank the admin team in UC CSSE who helped me through my research for every request or need I had.
- I want to thank IT CSSE team for their technical support and availability. Throughout my research period, I encountered a lot of IT challenging tasks and they were all dealt quickly and professionally usually within a few days, sometimes using their own private resources.
- I want to thank the student Adam Ros for his help in system testing as part of his summer project.

Contents

Part I

Background

Chapter 1

Introduction

1.1 Background and Research Motivation

Gathering data is a major obstacle for developing or improving algorithms in computer vision where the algorithms depend on statistical properties of the pixel values. For example, in image improvement methods, simulation cannot be a substitute for live data, since there is a strong connection between the algorithm and the actual problem domain. On the other hand, high-level algorithms which can be applied to various image understanding domains can be investigated and developed in simulated environments. Nowadays, game engines whose main purpose is to create games, have become more and more visually realistic in various animation domains which makes them possible candidates for providing visual data to computer vision experiments.

Research hypothesis: Game engine based simulation can increase the productiveness of computer vision algorithm development in a highly complex, natural and dynamic scenarios.

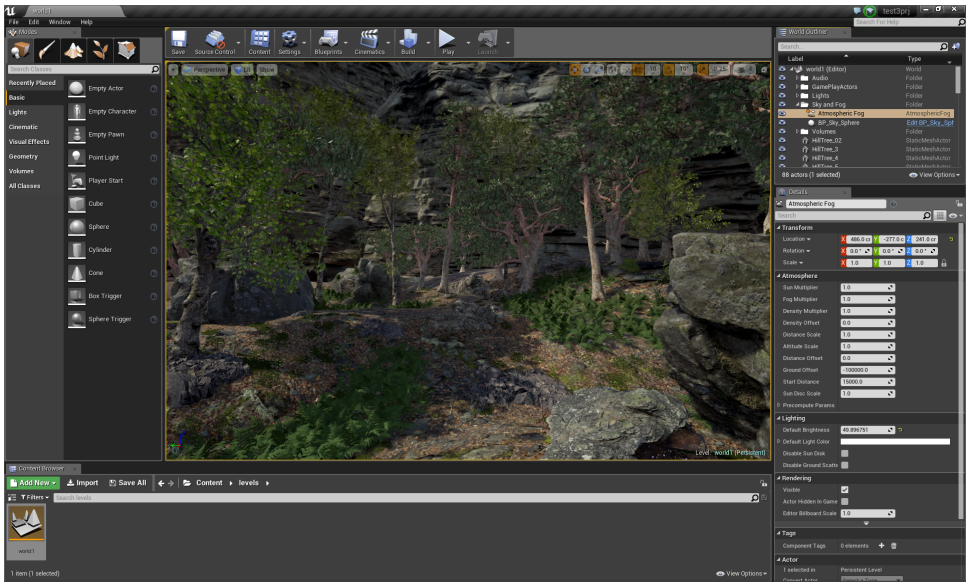
Research Objectives: The main objective of this research is to analyse the contribution and usefulness of simulation frameworks based on today's game engines for testing and validating computer vision algorithms for robots/drones/ROVs in highly realistic and complex environments such as underwater environments. Real computer vision problems from various challenging domains will be tested under the simulation and compared to real experiments. Several modelling aspects such as environmental effects, sensor data generation, controller characteristics and dynamics will be considered.

This research propose new solutions for different environmental domains in computer vision based on analysis and experiments done in the realistic simulation.

Our proposed methods enabled us to create research tools such as novel simulations frameworks with challenging environments for the use of the community in the effort of providing new tools for accessing new computer vision algorithm prior to real live experiments.

Importance of research: The advantage of using game engines as simulation engines can be enormous. Today's game engines can create almost realistic scenes (both visually and physically) and can be used to test image/vision algorithms in methods not available in the past. The development of game engine is growing side by side with robot and drone development. This presents an excellent opportunity for the development of autonomous robots in simulated environments, which can be used for all kinds of purposes, for instance creating a simulated disaster area and sending simulated robots as first responders trying to discover survivors or searching for a missing person in a vast simulated area.

Epic Games created a simulated world called "open world" [1] which can be used for that purpose. This emerging field can create a great scientific collaboration between game engine community, which includes artists, graphic designers, and developers to the rapidly growing robot/drone/ROV developers and user community. In fig 1.1 we can see simulated natural environment in the Unreal Engine 4 Editor and a real natural environment for comparison.



(a) Edited scene in Unreal Engine 4 Editor

(b) Real image

Figure 1.1: A complex natural environment modelled using UE4, and comparison with the image of a real scene.

Research Timing: Since this is combined with two rapidly growing fields (game engines, robot development), the development can continuously be updated due to ongoing developments in those areas.

Research Scope: We will focus on visually realistic real-time simulations created using a game engine. Realistic synthetic non-real-time videos can also be created by using Visual effects (VFX) tools like OctanRender [2]. These non-real-time tools can use slower more realistic methods such as ray-tracing and path-tracing to render frames. The game engine

(rasterization based rendering) will give us a more responsive real-time simulation which can interact with a user in a more real-time manner. In game engines, we will be able to run more experiments in a period of time compared to VFX tools. In addition, the echo system of a game engine enables us to create a packed game, a virtual interactive world which we can use and distribute as a platform for testing and collaboration.

We do expect that new hardware and better Artificial Intelligence (AI) techniques will catch up and will provide us in the future a more realistic renderers capable of real-time performance. Some of the latest advances will be covered in section1.3.

Advantages of using simulation In the past, it was hard to simulate relevant live scenes applicable to developing computer vision algorithms. A rendered scene will never look the same as a live view. However, the variety of scenes which can be created in a simulation is much greater and usually less resource-intensive to produce. Furthermore, some scenes cannot be created or tested in reality.

In the field of RF signal processing, simulation is a crucial and an accepted stage required for the development of systems and algorithms. We believe that today, due to the advancement of rendering engines and GPU power, similar methods can be applied to image processing and computer vision domains.

For the same scene, in simulation, we can change the conditions in which the algorithm is tested. We can shift from different lighting conditions like night and day, add wind which effects the vegetation, change the weather and so on. We can also analyze the contribution of each condition to the overall performance of the tested algorithm. In this work, we will cover these aspects thoroughly.

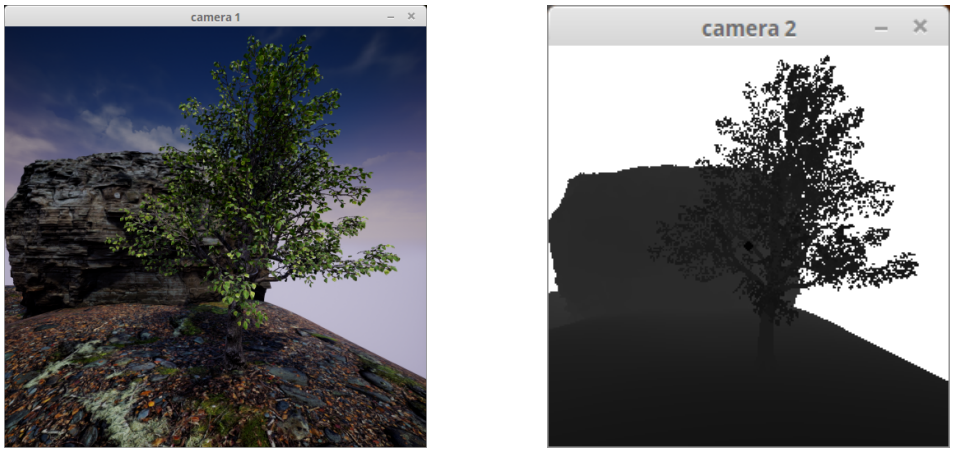


Figure 1.2: A scene image and the corresponding depth image

Another important use of simulation is in the creation of ground truth information, specifically in the area of computer vision. Getting accurate information from the simulation like collision and correct positioning is a vital part in navigation algorithm development. In figure 1.2 we can see a by-product of the rendering process which is the depth image which can tell us the distance from an object in the scene. Depth images can serve as ground truth information for testing algorithms like Simultaneous Localization and Mapping (SLAM) and can also be used to emulate a distance sensor like an ultrasonic sensor. Ground truth also helps in the development of monitoring tools, which are used to evaluate the performance in the simulated environment. Those tools can later be used to monitor real-life experiments.

1.2 Thesis Structure

This thesis is the product of comprehensive research into the field of visually realistic scientific simulation based on game engines. The first part focuses on the introduction and related work. This chapter provides the state of the art regarding the hardware, existing realistic simulations and their domain focus. We will provide related work on the area of realistic simulation and computer vision experiments. More related work material will also be given in context with the relevant presented domains in the following parts.

The next two parts correspond to two different environmental domains used for simulations: (i) drone in a natural environment domain (part II) in which we developed our main research tool the DroneSimLab. This domain is based on two published papers in addition to material that was not included in the original papers but is relevant to this research. (ii) ROV simulation in an underwater environment (part III) where we developed novel methods for vision based underwater autopilot navigation. This part is based on three published papers describing the process of generating a realistic underwater simulation with examples of experiments done in the simulation and in real underwater environments.

The last part IV describes ongoing research in the area of additional simulation domains, computer vision research and future plans. The last chapter summarises our research and conclusions. Addition work such as implementation details and general information regarding game engines will be given as appendices.

1.3 State of The Art

1.3.1 Hardware

2018 was an exciting year in terms of computer graphics. Nvidia started to support in the 20xx Turing architecture series Real Time Ray Tracing (RTX). Epic Games together with Microsoft declared that they will support RTX in DirectX 12 API and future versions of Unreal Engine [3]. From our perspective this counts as a major milestone in computer vision research since ray tracing is behaving more correctly in the way that light is interacting with the environment. The effects that were missing from the simulated environments such as reflected light from reflecting surfaces like water can have a huge impact on computer vision algorithms. Those reflecting surfaces will create mirrored features and mirrored objects. Alternative methods that were used in game engines such as screen-based reflections and for now still being used are a compromise for higher frame-rate and correctness. Screen-based reflections only mirroring the images and not taking in to account the right characteristic of the reflecting surface. In the case of water, that can be small waves and ripples. Up until now, only hybrid approach where possible due to the very high computational power needed for RTX [4] [5].

The new RTX capabilities where largely enabled by the entrance of AI into the computer graphics domain. Techniques such as AI denoising [6] enable to reduce the number of pixels needed to be rendered and the denoising filter manages to reduce the noise in a much more realistic way than traditional methods.

In addition to the RTX the Turing architecture enabled another computer graphics AI based technology. The Deep Learning Super Sampling (DLSS) handles Anti Aliasing (AA) in a rendered scene with minimal performance hit. Traditionally the AA was done by increasing the level of details being rendered which required more Graphics processing unit (GPU) power and more memory. The new Turing architecture is providing AI tensor cores for that purpose. This is another technology which is interesting in terms of computer vision since aliasing can create undesirable effects of non-realistic features, edges and noise.

Performance issues with the new RTX technology were accepted with mixed emotion by

the gaming community. RTX reduced frame rate sometimes by half from 100fps to 50fps when RTX was on. AMD for example, is still focusing on memory and memory bandwidth on their latest 7nm silicon-based Radeon VII hardware (Nvidia is still in 12nm technology) but they are also aiming for real-time ray tracing in the future, probably with less impact on the frame rate.

At the time of writing this thesis, there is still no support for RTX in the major game engines such as UE4 and Unity but we expect them to quickly catch up with the technology.

1.3.2 Scientific Simulators and Synthetic Databases

In the realistic simulator domain, Microsoft developed a framework to test drones [7] based on UE4 and is currently gaining popularity in the drone-related research. It is already being used for testing localisation techniques which is one of the most important tasks for drones [8].

The indoor environment is the type of environment where Global Positioning System (GPS) signal cannot be used for localization. MINOS [9] is a realistic simulator aiming to train agents to navigate in complex indoor environments. Zhang [10] used this simulator to train a navigation policy and then successfully applied it to a real environment.

In the driving domain CARLA an open urban driving simulator [11] was also used by Zhang [10] for the same purpose of training a navigation policy. CARLA was also used to create a resilient safety-critical automated system by testing all possible scenarios [12] something that is not realistic in real environment testing.

These simulation are all oriented to a specific domain mainly man-made urban spaces. Our work added a different approach in the form of a more generalised framework and additional environments as we will describe in later chapters.

Animation is also an area that is gaining more and more realistic capabilities. New AI methods such as Deep Reinforced Learning (DRL) are used to generate realistic robust animation [13]. The network is getting positive feedback by following the desired path. Getting feedback from the simulation reduces the need for large annotated databases as an input. We expect that in near future we will see more and more realistic character animation in computer games.

Mayer [14] analyzed how good synthetic databases are for computer vision tasks. Their finding shows that for simpler tasks such as optical flow and disparity estimation simple datasets with limited realism can be sufficient but might be a problem for higher level tasks such as object detection. This work increases the importance of using game engines for the low-level tasks which will be covered thoroughly in our research. Although we agree to their observation that the need for realistic scenes is overrated for the domain they worked on which is city spaces, natural scenes are more complicated and have less defined lines and corners in addition to sometimes more dynamic aspects due to the wind, leaves, falling snow etc. In our research, these realistic advantages will be highlighted and will give special attention to various simulation domains.

We can see that the research around synthetic databases is driven by popularity as we can see from the simulators created for the drone domain and it is also driven by industry interests such as the automotive industry. All the databases we encountered are based on urban scenarios. This is true not only for the synthetic ones such as Nvidia Drive [15] but also for real recorded databases such as KITTI and RoboCar [16] [17]. Direct interaction with game engines using our presented tools will be able to give researchers access to new types of domains as we will see in the following chapters.

1.4 Game Engines and Computer Vision Research

Some image processing and image-based algorithms have already been integrated into game engine/image simulation environments, although using game engines dedicated for games (like UE4, CryEngine, and Unity) in simulations is much less common. We believe the reason for this is that they are more focused on game experience as opposed to any real-world scientific applications involving simulations with associated mathematical and physical models and computer vision algorithms

An example of using scientific oriented simulation is the autonomous landing of a Vertical Takeoff and Landing (VTOL) Unmanned Aerial Vehicle (UAV) on a moving platform using image based visual servoing where they used Gazebo simulation [18]. SLAM is also tested and developed for indoor scenarios using Gazebo simulation [19] [20] [21]. Some environments are combined to create a more powerful engine, for example, Modular Open Robots Simulation Engine (MORSE) [22] combined with BGE (blender game engine) [23] and JSBsim [24] (an open source flight dynamics model).

In recent years, game engines have increasingly been used for simulations. Successful attempts have been made in evaluating the stability of structure using UE4, creating photorealistic scenes of stacks of blocks and applying deep learning methods [25]. A series of towers made from wooden cubes were created in a simulated environment using UE4 [26]. Some of the towers were stable structures, and some collapsed when the dynamic simulation was run. A network was trained to detect the outcome of the experiments. Testing the network on real environments achieved equal performance compared to human subjects in predicting whether the tower will fall. The most important aspect of this research is the fact that they could train the network on 180,000 scenarios which seem not feasible in a real-life environment.

A more recent work connected UE4 with OpenCV [27], the project is called UnrealCV [28]. It extends the UE4 with a set of commands to interact with the virtual world. Another work [29] proposed a new aerial video dataset and benchmark for low altitude UAV target tracking, as well as a photorealistic UAV simulator that can be coupled with tracking methods. Skinner [30] proposed a high-fidelity simulation for evaluating robotic vision performance for repeating robotic vision experiments under identical conditions. Similarly, one product of our simulation is a sandbox for high-fidelity simulations not only for algorithms but also for a full end to end Software In The Loop (SITL) simulations.

Another aspect is the increasing involvement of large organizations in robot simulation specifically and the release of their assets to the community. Microsoft developed a new real-world simulator to simulate drone in a real environment and released it on GitHub as an open source to the public [31]. This open source release is intended to be used by scientists to develop robots and drones in a realistic environment.

In some cases, companies are providing an SDK to communicate with the realistic simulated environment. SCS Software released an SDK for the Euro Track Simulator 2 [32] that allows developers and users to stream telemetry data from the game to any 3rd party applications. One exciting example for that kind of an application is the autonomous driving solution for the Euro Truck Simulator 2 [33].

Naturally, images created by game engines are still more oriented towards performance for the purpose of gaming. Tsirikoglolu compared his method of procedurally generated scenes to the existing state of the art synthetic world scenes such as SYNTHIA [34] [35]. Procedurally generated scenes might be a good approach to make the generated scenes more realistic and more versatile for training and testing computer vision algorithms. Since they used procedurally generated scenes, they assumed their method is 4 order of magnitude more efficient for generating scenes compared to the other databases.

1.5 Challenges

Using a simulated environment to train and develop an algorithm can present several challenges that are needed to be considered. Simulations tend not to capture the real problem or sometimes not giving the actual problem to the tested system, on the other hand, the simulation may present a problem to an algorithm which does not exist in a real-life scenario.

Using machine learning techniques, for example, can be risky in a simulated environment. In a simulated environment, the number and the variety of experiments can be significantly larger opposed to real life and can reduce the over-fitting problem. However, the trained machine can easily learn rendering artefacts which do not exist in a real filmed scene. Common rendering artefacts that can mislead an algorithm are popping effect due to changes in Level of Details (LOD) and aliasing in the rendered sequence. In our research, we tuned the game engine to avoid those problems as much as possible. For example, we use only one LOD on the expanse of a higher frame rate.

In image processing, feature detection algorithms working on a simulated projected image may also detect artefacts like aliasing. Anti-Aliasing techniques like filtering and supersampling should be applied in that case.

Another example of a possible pitfall: The game engine creates scenes that look genuine to the human eye (and brain) but not necessarily realistic from other perspectives. A world filled with vegetation is usually produced from a small sample of core objects positioned at different angles and sizes whereas in a real environment every object is different. This can mislead algorithms such as feature matching algorithms which rely on the fact that each feature should be different. One way to mitigate this problem is by using procedural generation of objects such as trees. We will describe this process briefly in later sections.

Considering these limitations, more traditional (non AI) computer vision algorithms are more suitable for testing under the simulation. Nevertheless, software stack based on AI can still benefit from realistic simulation by developing some parts based on real-world data and using the simulation as a completing method for testing additional scenarios and conditions.

1.6 Summary Of Contributions

Conference Papers

1. Ori Ganoni and Ramakrishnan Mukundan. A framework for visually realistic multi-robot simulation in natural environment. *arXiv preprint arXiv:1708.01938*, 2017. WSCG 2017 proceedings link

2. Ori Ganoni, Ramakrishnan Mukundan, and Richard Green. A multi sensory approach using error bounds for improved visual odometry. In *2017 International Conference on Image and Vision Computing New Zealand (IVCNZ)*, pages 1–6. IEEE, 2017 link

3. Ori Ganoni, Ramakrishnan Mukundan, and Richard Green. Visually realistic plankton models for simulating underwater environments. In *2018 22nd International Conference Information Visualisation (IV)*, pages 420–425. IEEE, 2018 link

4. Ori Ganoni, R Mukundan, and R Green. Visually realistic graphical simulation of underwater cable. In *Proceedings of the 26th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision, Plzen, Czech Republic*, volume 28, 2018 link

Journal Papers

1. Ori Ganoni, Ramakrishnan Mukundan, and Richard Green. A generalized simulation framework for tethered remotely operated vehicles in realistic underwater environments. *Drones*, 3(1):1, 2019 link

Software

The following software was developed as part of the supporting research tools for this thesis. All the software is open source and maintained in GitHub.

1. UE4PyServer- A Plugin for communicating between UE4 and the python programming language in the UE4 editor environment (Not actively maintained but can be used a reference) [41].

2. UE4PyhtonBridge - An extension to UE4 replacing the UE4PyServer suitable for both packed games and the UE4 editor environment. It is actively maintained and replaced the UE4PyServer [42].

3. DroneSimLab - The main framework simulation we developed for the purpose of this research extending the UE4 for end to end computer vision simulation. It is actively maintained supported and developed [43].

4. Turbulace model for ArduPilot [44] - DroneSimLab uses ArduPilot as a physics engine for the drone simulations. We contributed a wind turbulence model to ArduPilot rotating wing platforms. This model was needed for our drone simulations [45]. This code went through quality checks for acceptance.

5. UWCableComponent - An underwater cable simulation model was developed for the UE4 as part of our ROV simulations [46] [47].

6. OpenRov simulation model - for the underwater simulation we developed a dynamic model for the popular OpenROV the model is developed in an jupyter notebook and can easily modified for other ROVs [48].

7. RovVision - RovVision is a software and hardware framework for our future and current computer vision experiments. The repository include (i) ROV software, (ii) hardware designs (iii) Supporting documentation (iv) Visually debugging tools (v) Monitoring tools (vi) Simulation plugins to DroneSimLab (vii) Vision based stereo tracker (viii) Control algorithms [49]. This is an active project and continuously developed as main research tool of our underwater research team.

Hardware

This research included a significant hardware development and prototyping component as part of comparisons made between the simulation and live scene experiments. The following is a summery of the main components we used and developed.

1. Unsynchronised monocular sensor fusion rig - For a sensor fusion real experiment we used a Sony eye camera attached with 10 Degrees Of Freedom (DOF) Micro-Electro-Mechanical Systems (MEMS) device. In addition, markers where added to the rig to be used with an OptiTrex system for ground truth. The real experiment and materials are described in details in chapter 3.

2. OpenROV - The OpenROV platform was used in the initial stages of the underwater simulation research and was physically and dynamically modelled as described in chapter 7. The platform used as is with slight modifications to the tether system. We wanted to use the platform with minimal modifications in order to make the simulation a widely used tool.

3. Stereo Vision Rig for the BlueROV - The OpenRov ROV, although it was a useful starting point for the underwater research, it was not very manoeuvrable. it had only 3 thrusters compared to the eight we used in the BlueROV based platform. In addition we were able to extend the BlueROV platform both in terms of needed software tools and also by adding a stereo rig. We designed the rig for the purpose of doing underwater stereo based tracking experiments. The Platform is describes in chapter 8.

Part II

Drone Simulation Domain

Overview

We start this section with our research work on a generalized framework for the simulation of multiple robots and drones. The simulation architecture uses the UE4 for generating both optical and depth sensor outputs from any position and orientation within the environment and provides several key domain specific simulation capabilities. This simulation engine also allows users to test and validate a wide range of computer vision algorithms involving different drone configurations under many types of environmental effects such as wind gusts and lighting. DroneSimLab framework is an Open Source framework and was used for all of our simulations. As stated in the chapter it is actively developed and used by the drone community. Under this framework we explored and presented different tracking problems as practical test cases. In the next chapter we present another practical problem from the drone simulation domain which is the sensor fusion problem. In the paper itself we didn't use the realistic simulation but rather a more straightforward simulation approach of randomly created landmarks and adding noise to the landmarks positions. In chapter 3.5 we validate the results using the DroneSimLab framework and present our conclusion.

Chapter 2

A Framework for Visually Realistic Multi-robot Simulation in Natural Environment

2.1 Introduction

Graphical models of realistic natural environments are extensively used in games, notably simulation games and those that use immersive environments. These virtual environments provide a high degree of interactive experience and realism in simulations. Modern game engines provide tools for prototyping realistic, complex and detailed virtual environments. Recently, this capability of game engines has been harnessed to the advantage of computer vision community to develop frameworks that can be used in scientific applications where vision based algorithms for detection, tracking and navigation could be effectively tested and evaluated with various types of sensor inputs and environmental conditions. This chaper focuses on the development of a comprehensive standalone framework for multi-robot simulation (specifically, multi-drone simulation) in complex natural environments, and proposes suitable configurations of tools, simulation architectures and also looks at key performance issues.

Several robot simulation engines exist which simulate different robots and vehicles e.g. multicopters, rovers, fixed wing UAV, etc. Each engine has its advantages. The engines use large simulation environments consisting of models, sceneries, etc. generated by other simulation packages and frameworks. Following are some examples of such engines with dependencies on other simulation packages:

- Standalone robot simulation engines using a flight simulator for models, sceneries and functions for visualization and simulation. Examples of such packages are: (i) ArduPilot [50] which communicates with Xplane [51] and Flightgear [52] (ii) PX4 [53] communicates with jmavsim [54]. Flight simulators are usually much larger projects than robot simulation projects. They are more focused on user experience and interaction, but they also have visualization and dynamic simulation capabilities which are useful characteristics for drone projects.

- Standalone simulation packages that use physics engines, graphical interfaces and simulation capabilities provided by other simulation tools: for example PX4 [53] with Gazebo [19]. Robot simulation environments are dedicated simulation environments. They are focused on giving proper tools for modeling and simulating robots but are less focused on visualization.

- Stand alone robot simulation environment: those environments include the robots and

flying vehicle models. An example of that kind of environment is: MORSE. Those environments are suitable for testing and evaluating ideas, but they don't have roots in real robot projects specifically in drone projects.

- Game engine stand alone environment: the robot is simulated inside a game engine. For example a benchmark for tracking based on UE4 [29]. Similarly to robot simulation environment, the drones inside game engines don't have roots in real drone projects. Additionally drone simulated in game engines don't share the dynamics of real drones. For example, they don't have to deal with wind gusts and vibrations.

In this chaper, we propose a novel configuration that use game engines for the simulation environment, the primary motivation being the enhanced capabilities of a game engine such as UE4 in providing highly realistic environments and various modes of visualization. One of the primary advantages of this type of a configuration is that a game engine such as UE4 can provide realtime videos of camera output based on the position and attitude information of the robot. This chapter also gives an overview of the DroneSimLab [43] developed by us, which has constantly evolved with the analysis of various requirements and concepts related to the simulation architecture presented later in this work. The design and implementation aspects of the key components of this simulation engine have been presented in detail.

This chaper is organized as follows: in section 2.2 we give an overview of the DroneSimLab project. . Section 2.3 gives detailed information about the framework simulation architecture and design goals. Section 2.4 focuses on modifications needed to be made to meet the simulation design goals. Section 2.5 describes experimental results and performance. Section 2.6 provide information on future research directions as well as references to online demos of this research.

2.2 The DRONESIMLAB Project

We developed DroneSimLab as an opensource project to foster collaborative development of drone simulation packages that use the power and capabilities of the UE4 as discussed in the previous sections. Some of the main functionalities which the current implementation provides are:

- Multi-robot - can handle more than one robot and create visual interaction.
- SITL driven - can simulate two drone models: ArduPilot and PX4
- Based on Game Engine - Uses UE4 as an optical and depth sensor
- Realtime - depends on the hardware but can run at 30 fps.
- Natural environments - can simulate trees, wind grass, etc. (comes with Game Engines assets).

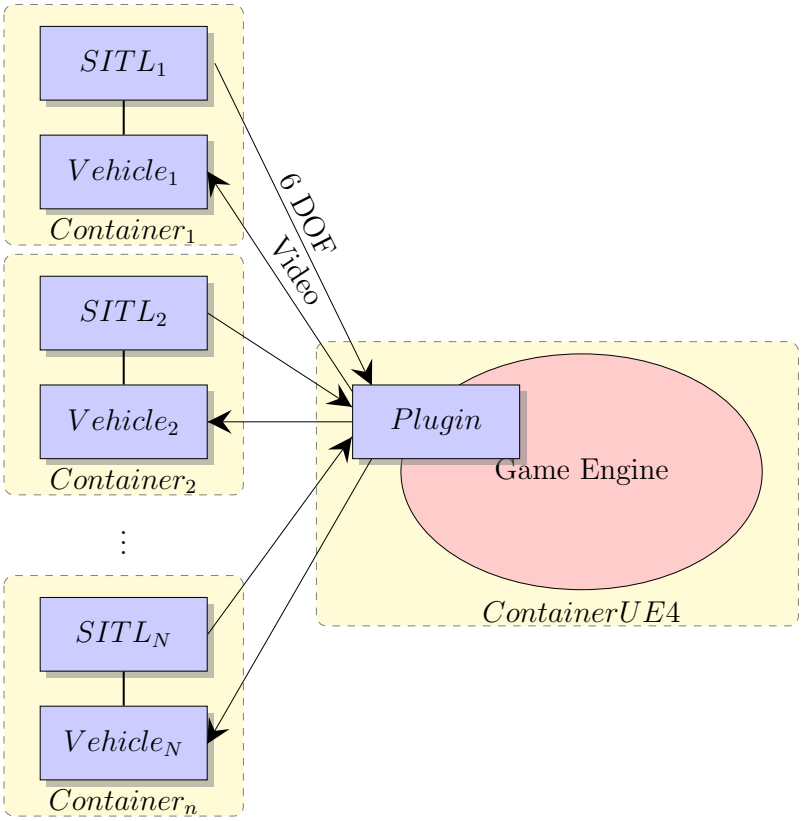


Figure 2.1: Simulation architecture

2.3 SIMULATION ARCHITECTURE

In this chaper, we propose a simulation architecture designed to meet the following primary goals: (i) ability to generate realtime camera outputs for any arbitrary position and orientation in a natural environment, (ii) ability to integrate software and hardware in the loop simulations (iii) ability to combine multiple simulations and (iv) ability to reproduce results. These aspects are elaborated below.

2.3.1 Domain Specific Simulation Engine

We focused on three simulation engines for the framework.

- The game engine provides video, depth data and additional visual environmental effects like wind and dust.
- The physical model engine, usually supplied by the robot development framework.
- Supplimental simulation objects like communication channels models , computation power restrictions and additional simulation filters (for example a lens distortion filter).

Engines can create an environment for a single robot. For instance in the case of SITL, the simulation engine interacts with only one vehicle and produces sensory information for only one robot. On the other hand, the game engine can provide visual information for multiple robots as described in Figure 2.1, this is especially necessary if a visual interaction exists, for example; one robot can block the field of view for another robot, and this aspect should be implemented in the simulation. By embedding SITL into our framework we ensure that the simulation is highly correlated with the real robot architecture (since it is used for the robot development). Hardware In The Loop (HIL) can be later used for further validation.

2.3.2 Simulated sensor architecture

We identified three types of simulated sensors that can be used.

- Single domain sensor - lives only in one engine. For example a simulated RGB camera from UR4 [26]. Another example is the gyro sensor, which is simulated only in the SITL software e.g. gazebo, jmavsim, jsbsim etc.
- Multi domain sensor - lives in more then one engine. For example such a sensor can be seen in Figure 2.2. In this example the simulated distance sensor gets information in from various sources like an external Digital Elevation Model (DEM).
- Complex sensor - lives in both the physical domain and in the simulated domain. An example of such sensor is a camera in front of a screen. The display provides the visual information and the camera is used just as in the real system, enabling monitoring real system performance and hardware issues. This concept is an extension of the HIL mode which combines hardware testing and software testing.

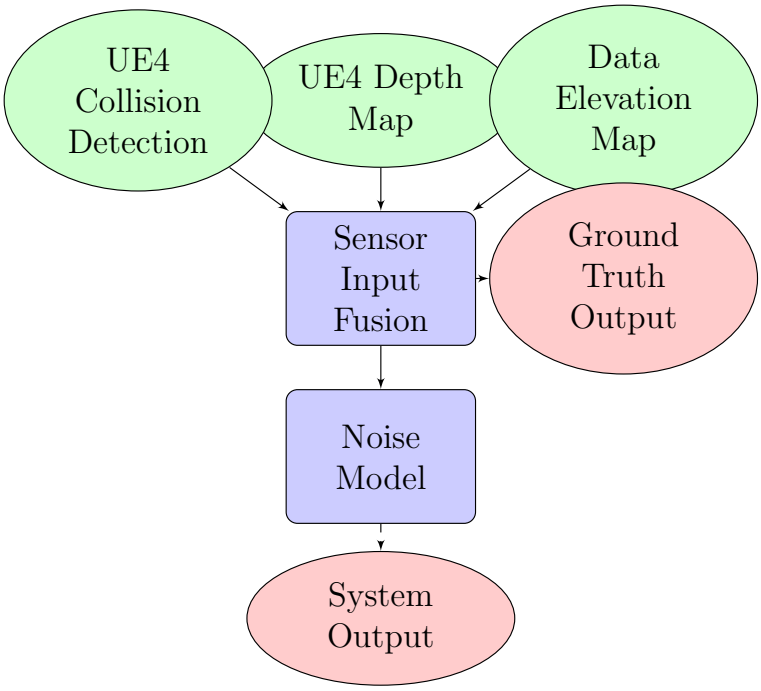


Figure 2.2: Example of a multi-domain distance sensor architecture. The simulated sensor accepts inputs from different engine resources and produces ground truth information and noisy output.

2.3.3 Containers as vehicles

We used a container approach to combine simulations not originally intended to work alongside each other. The container approach enables us to use different operating systems and libraries on the same machine. We can also control the network configuration in each container. This approach is different from other frameworks like AirSim [31] by maintaining the original firmware developed by the Robot Simulation Framework. Our architecture can utilize the benefits of new features and continuing development of those environments.

2.3.4 Reproducibility

The ability to reproduce results under differing or constant conditions is vital in system development as well as in research, and becomes more and more difficult as the complexity of the

system increases. To realize this concept, we are using several existing software tools. We are using the Docker engine to manage system configuration, and Git version control to handle the software development. Since the experiments are in a simulation, other reproducibility aspects such as high-fidelity are built-in. In section 2.5, we demonstrate testing of algorithms in complex natural environments by controlling simulation parameters. In this simulation, we can see that outdoor natural environment can be problematic for testing visual algorithms since we don't have full control of the environment. It seems that true reproducibility in an outdoor natural environment may be achieved only in simulation [30].

2.3.5 Build system & configuration management

The simulation environment uses these software tools:

- Version Control - All files of this project are managed by Git version control under GitHub servers [55]. The only exceptions are the UE4 projects which are managed locally due to the large file sizes. The UE4 [26] source code is still managed by git in dedicated GitHub repository. For the purpose of sending realtime ground truth position, changes have been made both to ArduPilot Project and to PX4 and are managed in separate forks. Those changes are not compatible with the design and purpose of the original projects. Changes that were compatible (e.g. a turbulence model) were returned to the community as pull requests and then pulled back into our local fork.
- Containers - Created with Docker engine.
- ArduPilot Fork [50] (Drone Project)
- ROS - Supporting firmware for the PX4 project.
- PX4 Fork [53] (Drone Project)
- UE4PyServer plugin [41]

2.4 Engine Modifications

2.4.1 UE4 Plugin

Game engines are not dedicated research tools, obviously, but conveniently for our usage scenario they supply mechanisms like plugins to extend the capabilities of the engine. The plugin we used for the UE4 [26] is called UE4PyServer [41] Plugin and was developed for the purpose of this research. The main concepts behind the plugin development were:

- Realtime: For this simulation, we took advantage of the realtime capabilities of the game engine. Realtime simulation (RT) is important when you want to run many tests in a short period. RT simulation is also necessary when human interaction is involved because users expect realtime or near realtime behavior. To maintain RT behavior, the UE4 plugin was developed with minimal processioning on the UE4 side. The primary purpose of the plugin is to communicate with other parts of the simulation. e.g. receiving 6 DOF information and sending video data.
- Multi-Robot support - UE4 enables capture of the viewable screen to a file or a buffer, but this provides us with only one camera feed. To allow multiple cameras in the simulation, we used rendering-to-texture technique with object ScreenCapture2D [56]. The method

is used in the game engine usually to render surfaces like security cameras, billboards, mirrors, etc. We used it to simulate a camera robot and depth sensors using the depth map provided by the ScreenCapture2D Object.

- Synchronization - We wanted the sampling to be synchronized for all the visual objects in the simulation. It is an important concept and might be critical for some applications, for example, simulating stereo camera. For this purpose, we used coroutines which are a light version of synchronized pseudo-threads.

2.4.2 Building realistic environment inside game engine for computer vision

There are some special considerations when building virtual environments in game engines for computer vision purposes.

- LOD - in game engines using multiple LOD [57, Chapter 3] in order to maintain graphics performance especially frame rate. This may create unnatural textures changes which can be destructive for computer vision algorithms. For the purpose of this research, we can control the environment and the simulation, and we can use that to create a scene with only one LOD.

- Repeating patterns - In Figure 2.3 we can see the meshes used to build the realistic scene. To reduce the effect of repeating patterns, each element is positioned in a different orientation and slightly different scaling. Also, the elements are positioned with some overlap with other objects which reduces the repeating effect.

- Culling adjustments - the area rendered in the scene also known as frustum should be large enough for all the objects in the scene to be rendered, so we will avoid popping effects due to movements of the cameras or the objects themselves.

- Dynamic shadows adjustments - moving objects in the scene like trees and robots should always cast dynamic shadows to imitate real scenarios.

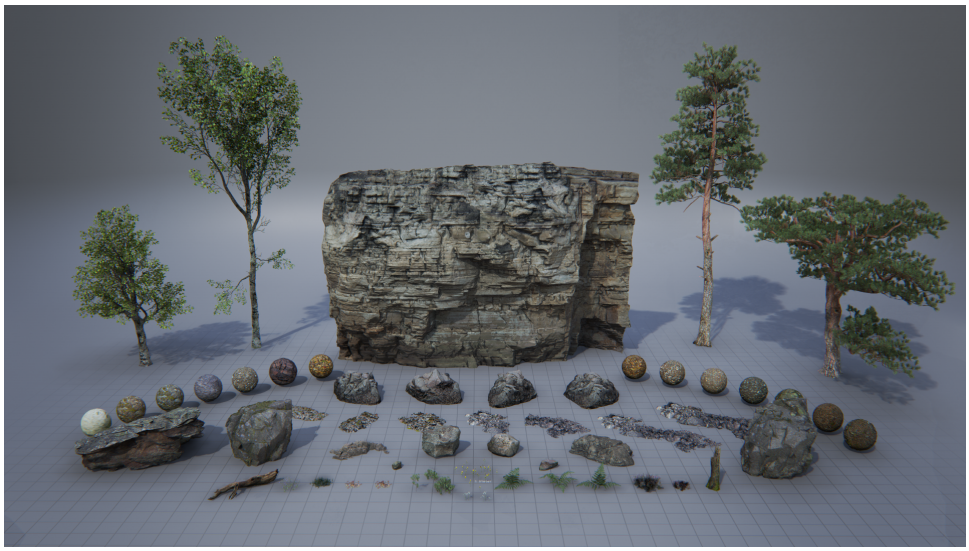


Figure 2.3: Open world Overview - these are the assets used to build a forest environment

2.4.3 SITL

The SITL engine needs to send 6 DOF information at a high rate to the game engine (at least 30 fps) to maintain realtime constraints. For that purpose, some modifications are needed to the engine, so the SITL engine will send ground truth information directly to the game engine, and also to logging mechanisms for later analysis as described in Figure 2.1.

2.5 Experimental Results and Performance Analysis

2.5.1 Plugin tests in natural environment

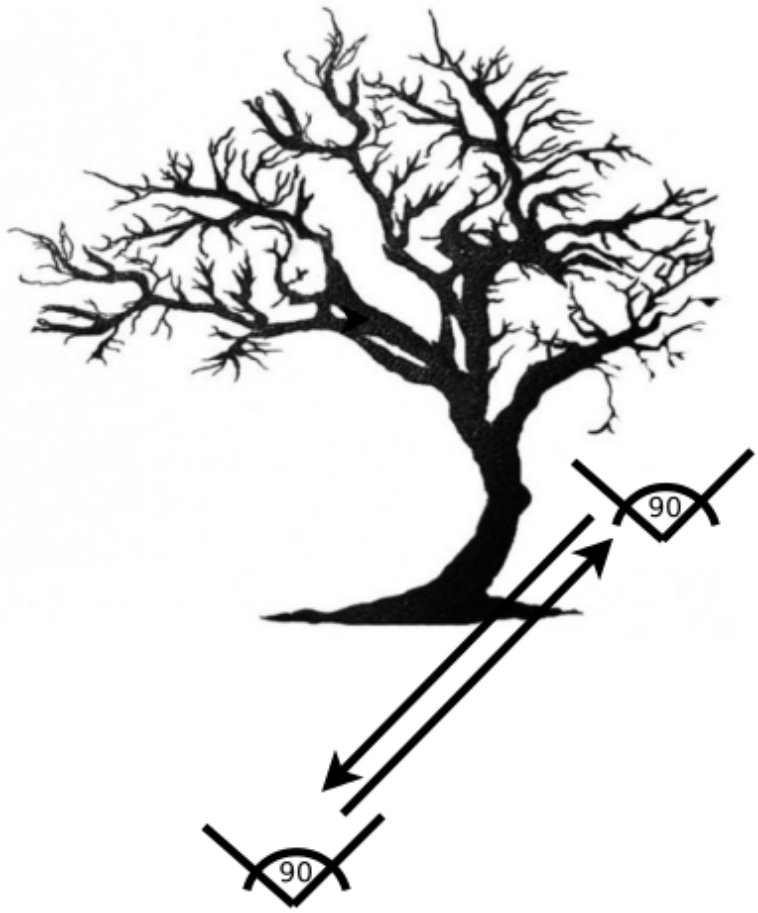


Figure 2.4: Scene architecture - In the UE4 Editor, we placed a camera (with 90° Field Of View (FOV)) at an initial position, in front of a tree. We moved the camera diagonally away from the tree (going backwards and to left wrt to the tree) , and then returned to the same point. Camera maneuvers which starts and ends in the same position are ideal for tracking tests. Ideally, the tracked points should get back to the same original coordinates.

Running visual algorithms in a natural environment can be very challenging. Relative to artificial environments, natural scenes can by highly dynamic due to atmospheric conditions such as wind, and usually will not have distinct characteristics like straight lines, circles corners, etc. Using UE4PyServer [41] (which was developed as part of the simulation framework) and UE4 [26] we developed a tracking simulation (live video can be found here [58]) to demonstrate the uniqueness of natural environment. The simulation is based on the Lucas-Kanade Optical Flow tracker implemented in the OpenCV library [59] which we use it to track an ordered grid

of points (no feature extraction). The maneuver is a simple camera facing forward and moving diagonally back and then return to the original position as described in Figure 2.4. Ideally, we would expect that the tracked points will return to the same coordinates when the simulation cycles back to the starting frame. Since this is a complex 3D scene, not all the points will return to the same location due to the loss of tracking, but in Figure 2.5 we can see that running the experiment twice produces similar results. Similar but not exact, since there is still some randomness in the scene due to movement of leaves that might cause slight differences. In Figure 2.6 we conducted two experiments with the same setup, but in the second test, we add the wind to the scene by adding to the UE4 a Wind Direction Force object. We can see that the results are now very different. We repeated the experiment under various conditions and calculated the following MSE grade to quantify the tracking quality:

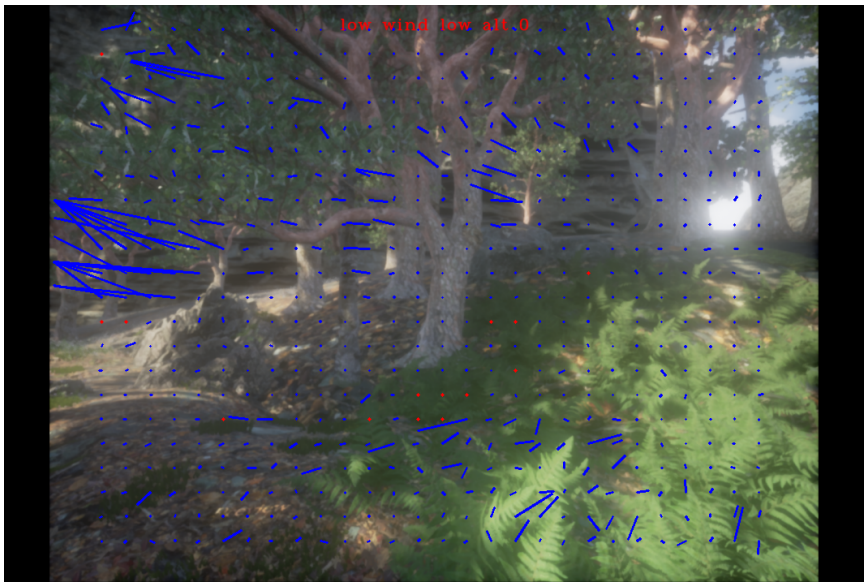
$$G = \frac{1}{N} \sum_{P \in T_p} |P_e - P_s|^2 \tag{2.1}$$

where: G is the tracking error, T_p is the tracked points, P_s an P_e are the start and end coordinates of the tracked points and N is the number of tracked points. Summarized results are in the following table:

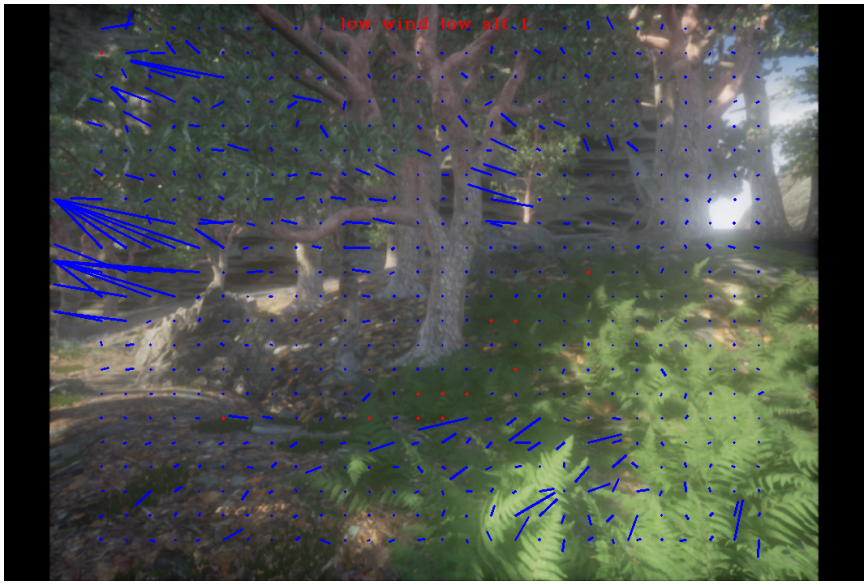
	low-wind	high-wind
low-alt	96.3 (98.54%)	219.9 (97.40%)
	81.1 (98.54%)	225.4 (98.05%)
	87.6 (98.54%)	234.4 (97.89%)
high-alt	55.7 (97.56%)	243.5 (98.54%)
	53.8 (98.21%)	757.6 (97.73%)
	48.9 (97.40%)	382.7 (98.38%)

Table 2.1: Tracking error (MSE) values in pixels^2. The numbers in brackets give the percentage of points that the Lucas-Kanade Optical Flow tracker [59] marked as valid.

In Table 2.1, we can see the performance of the tracking algorithm under different environmental conditions. As expected in high altitude (near the tree tops) with the combination of strong wind will be the most challenging scenario. As seen in the first column, the tracking error under low wind conditions is larger at low altitudes compared to high altitudes due to the presence of a higher density of objects such as leaves and branches that occlude the camera view at low altitudes. On the other hand, when the wind speed increases, the trend is reversed because leaves and branches tend to move more than the objects closer to the ground like tree trunk rocks, etc.



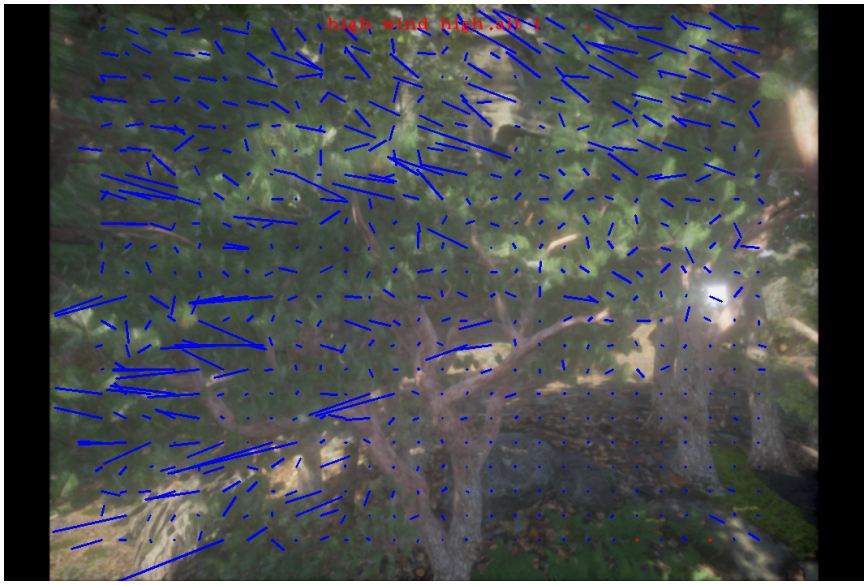
(a)



(b)

Figure 2.5: Simulation outputs (a) and (b) showing reproducability of results under similar environmental conditions. The blue line connects the initial position of the grid features with their final positions. Since the camera returns to the exact same point, ideally we should expect to see a blue dot and not a blue line which represents some error in tracking. Red dots represent points the tracker was unable to track

(a)



(b)

Figure 2.6: Simulation outputs showing the variations in results under differing environmental conditions. The output in (a) was generated without simulated wind, while in (b), wind was added to the simulation. We can observe degradation in the tracking quality of the grid features.

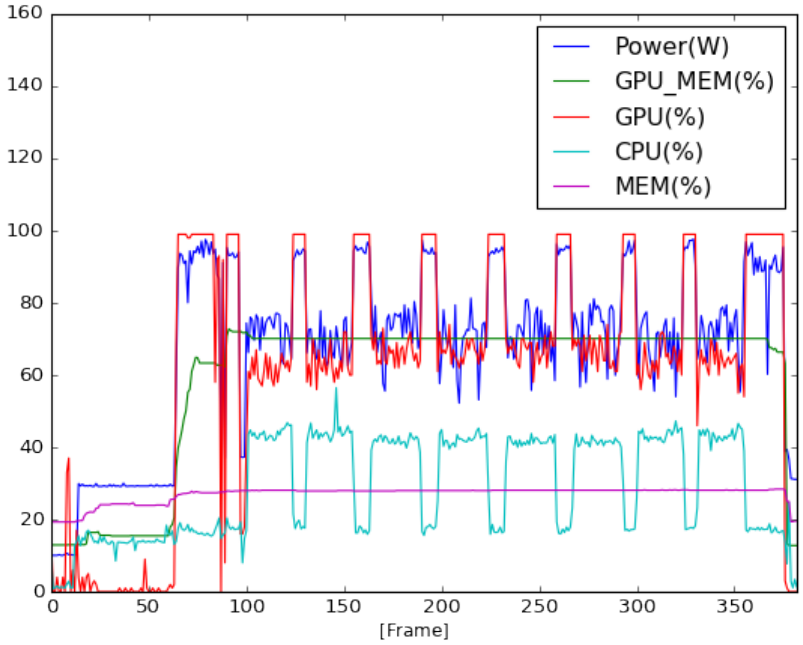


Figure 2.7: Profiling scene - high level profiling during scene playback created with "nvidia-smi" command line tool and Python "psutil" library.

We developed a tool for profiling and monitoring the framework in addition to the existing tools in the UE4 editor. This is a high-level profiling tool that gives us the summary of the system utilization. An example of a test case is presented in Figures 2.4 and 2.7. Fig 2.4 explains the scene architecture and Figure 2.7 the corresponding profiling graph. The peaks in the GPU utilization are due to camera IO-intensive movements as would be expected. When the camera was moving, we observed that the GPU is fully utilized (it reaches 100%) resulting in reduced framerate and when the frame rate is reduced the CPU was less occupied because it was processing the images at a lower frame rate.

2.5.2 DroneSimLab tests

We created a setup in DroneSimLab for an experiment of one drone tracking another drone. The drones are Ardupilot drones simulated using their internal SITL engine. We simulated the wind in the UE4 as well as in the SITL engine including wind gusts. The two drones fly into the forest and then return to the original position [60]. One of the drones is using HSV tracker [61] to track the other drone (Figure 2.9). We repeated this experiment four times and the results are presented in Figure 2.8.

In all four scenarios, we can observe the loss of tracking capabilities when the drones enter the forest (black dots between frames 300 to 500 in all four scenarios). When the drones enter the woods, the shades from the forest canopy affects the color and brightness components of the drone as can be seen in this demo video [60]. The threshold for tracking is not updated dynamically to demonstrate this behavior. Other interesting phenomena is the high frequencies observed in the graph. These high frequencies result from the continuous maneuvering and changes in the 3D orientation of the drone to compensate for the high wind forces, which in turn results in variations in the estimation of the drones center position. In the last two experiments, we can see especially large amplitude in the beginning and at the end of the experiments as a result of the takeoff and landing process which provided different angles of viewing of the drone body led to a different estimation of the drone center.

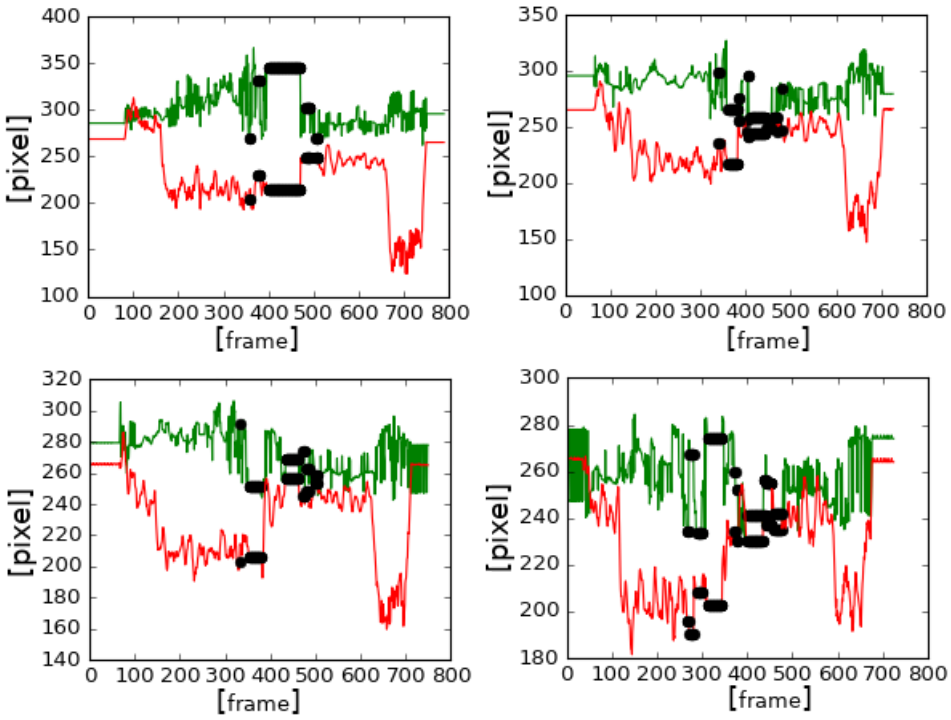


Figure 2.8: Four consecutive tracking experiments results. The black dots represent tracking failures. The X axis is frame number and the Y axis is pixel position. Green and red lines are the X,Y pixel coordinates respectively.

The above results have clearly demonstrated the usefulness of a game engine in not only producing realistic natural environments and their camera outputs, but also providing the ability to add and modify realistic environmental effects such as changes in wind parameters and illumination conditions. These features allow us to generate ground truth data for various test conditions and to evaluate machine vision algorithms.

Figure 2.9: Tracking experiment - A drone following another drone in DroneSimLab.

2.6 CONCLUSIONS

The creation of visually realistic environments is a very powerful tool for computer vision research as can be seen in section 2.5 and this corresponding video demo [58]. The DroneSimLab project [43] aims to be a tool which adds game engines capabilities to the current existing robot simulation environments. The current work mainly focuses on UE4, but adding another game engine may increase the dimensionality of modifiable parameters in our systems. For instance, training deep learning algorithms on multiple worlds each created by a different game engine may more accurately generalize to the real world domain. This chapter has presented a new framework for simulating multi-robot (specifically, multi-drone) motion in such environments, where environmental effects can be easily incorporated, and complex computer vision tasks evaluated. The simulation architecture along with the key functionalities of the simulation engine have been discussed in detail.

Chapter 3

A Multi Sensory Approach Using Error Bounds For improved Visual Odemetry

3.1 Introduction

Visual Odometry (VO) algorithms play an important role in estimating the ego-motion of autonomous vehicles such as mobile robots and drones. In computer vision, a lot of research is currently focused on localization algorithms such as SLAM. For example, a robust monocular SLAM algorithm was recently proposed in [62] and [63]. Those approaches suffer from the lack of continuity in the solution. Loop closure detection in SLAM is also a challenging problem, which if not executed correctly can be destructive to a control process. For instance, if the robot enters the wrong room and the loop closure process detects it and updates the robot position, that kind of jump can create unwanted behavior of the robot to correct its position.

One of the current research topics in visual odometry using multiple sensors is the use of inertial measurements together with vision information as proposed by Rovio [64]. The tracking of the multilevel patch features is closely coupled with the Extended Kalman Filter (EKF). In their paper, they used a dedicated hardware which created a synced stereo camera with inertial sensor measurements. Leutenegger [65] formulated a rigorously probabilistic cost function that combines reprojection errors of landmarks and inertial terms. Their solution used a bounded window of keyframes and an optimization of the reprojection errors of landmarks. Their paper presented a comparison of both a stereo and monocular version of their algorithm with and without online extrinsics estimation with respect to ground truth. They used custom-built stereo visualinertial hardware that accurately synchronized accelerometer and gyroscope measurements with imagery. Another hardware based solution was suggested by [66] for a fixed wing UAV. These implementations deal with the continuous case using the EKF and the bounded window.

In this chapter, we present an experimental setup and results obtained using low-cost unsynchronized hardware and a per-frame optimization of the reprojection error. This kind of software based approach is applicable for a lot of robotic applications where the attitude information changed relatively slowly and is available through various sensors. This problem is usually solved by higher level algorithms which fuse the inputs from various sensors to a unified solution. For example, it may ignore the visual odometry sensor output if it is inconsistent with other sensors. In this chapter, we discuss an experimental test case that uses off-the-shelf low-cost hardware and show the advantages of fusion of image odometry sensors and inertial sensors as a first step in an autonomous vision-based navigation pipeline (e.g. before the EKF stage). We also evaluate the performance of the proposed approach using computer simulation and a real world experiment done in a controlled environment.

This chapter is organized as follows: The next section presents an overview of our proposed

algorithm. Section 3.3 describes in detail the simulation and the experimental setup. Section IV provides a detailed description of the experimental results including comparative analysis and performance evaluation. Section V concludes the chapter with a summary of the important concepts and results presented, and also outlines future work.

3.2 Algorithm Overview

In this section we provide an overview of the proposed algorithm for improving the accuracy of visual odometry measurements using error bounds and information derived from multiple sensors and apply it to a test case scenario which will be later tested in simulation and real environments. The algorithm is divided into four main steps as shown in the block diagram in Fig. 3.1. The first three steps are designed to establish a baseline reference between two camera views (points B, C in Fig. 3.2) for the purpose of tracking and triangulating features in order to extract 3D landmarks. The fourth step is a continuous step of recovering the 3D camera orientation and position based on the landmarks and the tracked features using the SolvePnP function in OpenCV [27], from this point we will refer to it as SolvePnP1 . In this step the camera moves along a rectangular path (points D,E,F,G,H in Fig. 3.2).

One of the fundamental equations used in visual odometry is the following equation that defines the epipolar geometry between two views:

$$[P_2|1]^T K^T E K [P_1|1] = 0 \tag{3.1}$$

where P_1 and P_2 are the corresponding 2D points matrices obtained from the images of the landmarks in the two views, K is the camera matrix used by both positions and E is the essential matrix. The "findEssentialMat()" function in the OpenCV library provides the essential matrix, and the "recoverPose()" function is used to extract the rotation and translation vectors using SVD decomposition [67]. It should be noted here that the relative translation between the two camera views is specified by only a unit vector, and therefore known only up to a scale. We can update the translation vector to the correct scale using measurements h_d from an altimeter, as shown below:

$$\hat{C}_d = -R^T \hat{T} \tag{3.2}$$

$$S = h_d / \hat{C}_{d2} \tag{3.3}$$

$$C_d = \hat{C}_d \cdot S \tag{3.4}$$

$$T = -R C_d \tag{3.5}$$

where R is the rotation matrix and $\hat{T}$ is the unit translation vector returned from the "recoverPose()" function in OpenCV. The term S represents the scale factor calculated using the unit camera translation vector $\hat{C}_d$ and the actual change (delta) along the direction of the altitude vector given by h_d. The altimeter sensor gives an estimate of this vertical offset h_d. $\hat{C}_{d2}$ is the altitude component of the camera's normalized translation direction vector. C_d represents the camera's position delta after scale correction. T is the translation vector after scale correction.

The next step is to find the camera position and orientation. The SolvePnP function in OpenCV implements Zhang method [68] and is used to estimate the camera pose from 3D-2D correspondences. We run the this function continuously with the solution in the previous iteration used as the guess for the next iteration. We developed a modified version SolvePnp2 of the above function where error bounds are used to obtain solutions of a bounded minimization problem as given below.

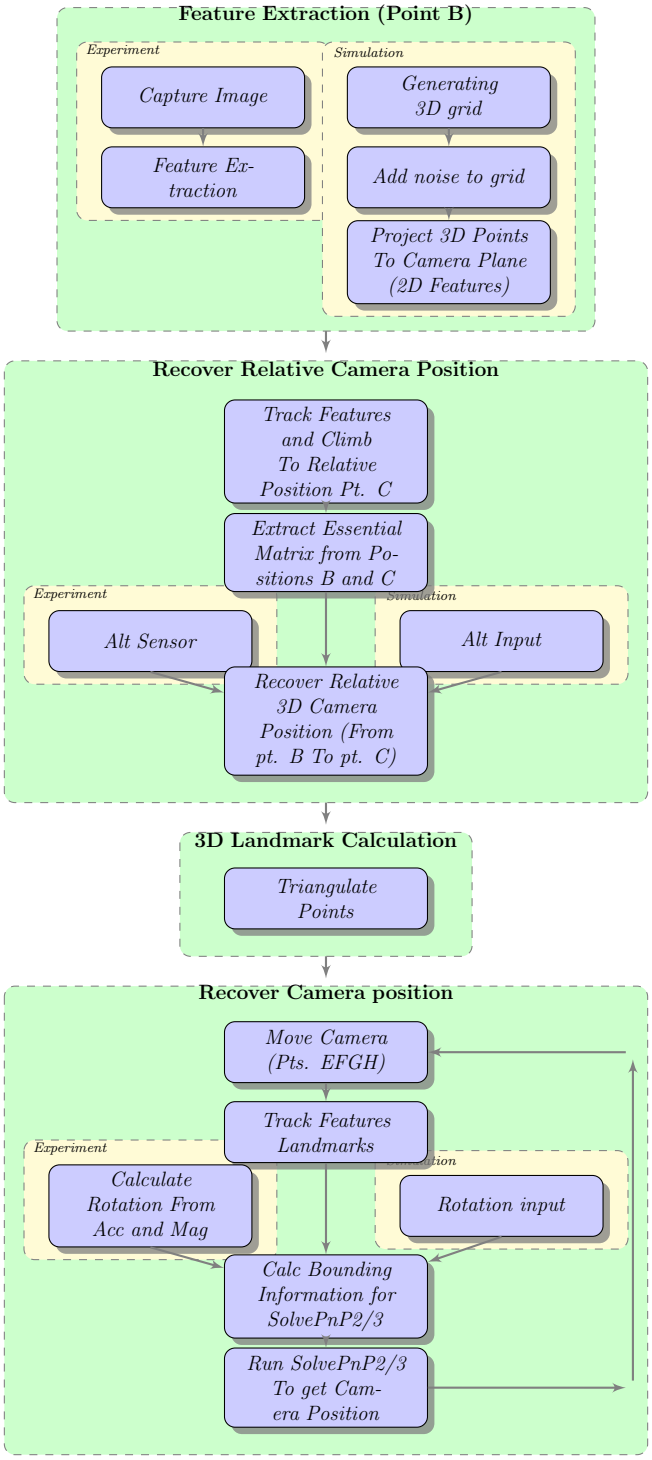


Figure 3.1: Algorithm Overview Block Diagram

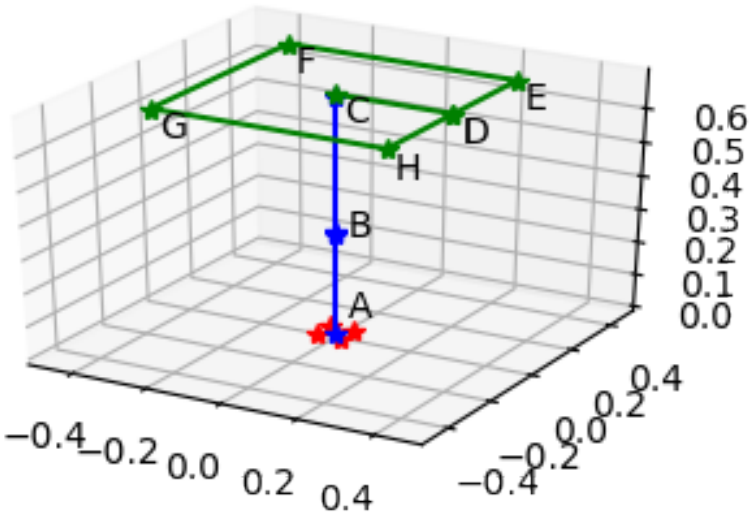


Figure 3.2: Camera Course: The camera starts at a slightly elevated position (B) facing downwards to the features (A) and continues to the second position (C) to extract a relative orientation, continuing from there the the repeated rectangular course ($EFGH$)

$$\begin{aligned} \underset{X}{\text{minimize}} \quad & \mathrm{F}(X) \\ \text{subject to} \quad & X_i \in (l_{bi}, u_{bi}), i = 0, 1..5 \end{aligned} \tag{3.6}$$

where F is the cost function and X is the optimized vector, X_0, X_1, X_2 represent the camera orientation in both implementations of SolvePnP and X_3, X_4, X_5 represent the camera position to be optimized. l_{bi}, u_{bi} represents the applied lower and upper bounds respectively. In our case, we applied boundaries to X_0, X_1, X_2 (the orientation) and for some cases we bounded the value of X_5 which denotes the altitude of the camera. The applied bounds depend on our estimations and assumptions regarding the additional sensors accuracy and a priori knowledge of the system state. The cost F in our implementation of the SolvePnP2 function is calculated as follows:

$$\begin{aligned} R &= rodrigues(X_0, X_1, X_2) \\ T &= -R \begin{bmatrix} X_3 \\ X_4 \\ X_5 \end{bmatrix} \\ Cost &= \sum_{i=1}^{n} |(RV_i + T) - P_i|^2 \end{aligned} \tag{3.7}$$

where X is the optimized vector, R is the rotation 3x3 matrix extracted using the Rodriguez formula to convert axis angle to rotation matrix, T is the translation of the camera, V is the 3D position array of the extracted features (landmarks), P is the 2D tracked feature points on the image plane. The decision to use the camera position and not the translation vector T for the state vector X was done intentionally in order to enable altitudes bounding where altitude information exists. That information can come from altitude sensor or in ground robot may be known to some amount of certainty or it can even come from an output of higher level navigation algorithms like EKF [69].

SolvePnP2 uses the angle axis representation which is used widely in optimization problems since it is a compact 3 variable representation and it corresponds to the 3 degree of freedom of the 3D rotations. The downside of this representation is that it is difficult to have physically reasonable bounds on the 3 axis angle vector. For instance if we limit the vector to: $l_b = -\theta, u_b = \theta$ (lower and upper bound) these two vectors $(\theta, 0, 0)$ and $(0, \theta, 0)$ are both

legitimate under the bounding conditions but are actually very different in terms of pointing directions. We therefore created another version of the function SolvePnP3 that uses Euler angle representation. Euler representation can be better fitted to real life situations where different sensors are responsible for different angles for instance the accelerometer is responsible for the pitch and the roll angles, while the magnetometer can be more related to the yaw angle in most cases. In the Euler representation we need to consider the non continuity of variables representing angles, for example at 360°. In the Euler representation we have three angles (opposed to only one in the axis-angle representation) which may prevent convergence. We can solve this issue by bounding the angles to avoid the singularities areas and use relative rotation as opposed to absolute rotation, as relative rotation can be in some cases small or around zero angles.

3.3 Methods and Tools

3.3.1 Optimization Software

We used least_squares function from the Scipy optimize package [70] for running the least square optimizations for our modified version of the SolvePnP function. The function was activated in the Trust Region Reflective (TRF) mode. The algorithm is quite robust in unbounded and bounded problems and has similar comparable results in an unbounded environment to the SolvePnP which makes it ideal candidate for our experiments.

3.3.2 Simulation

We developed a simulation system to get high accuracy ground truth data and maximum control over the tested environment regarding added noise. A simulated camera was created by projecting a grid of points on the simulated camera plane based on the camera position orientation and the camera matrix. We added normal noise with standard deviation of 1cm in each axis to each of the extracted landmarks. The noise was added to the simulation process after the triangulation. This was done to create a meaningful error regarding accepted accuracy in triangulation along the Z axis (the height of the camera). The error addition in this stage represents the problematic behavior of computer vision applications. In a discrete point in time, a wrong estimation can occur which in turn will have long term effects on the system navigation and control capabilities. A random seed was added to the simulation to make it repetitive and to enable accurate comparisons between methods. We run the experiments with and without the added noise for comparison.

Another aspect of the simulation is the simulated maneuver. The chosen maneuver was as described in Fig. 3.2 but the rectangular part of the maneuver was repeated twice; once without changes in the orientation of the camera and again with tilting the angles along the course as shown in the left sub-figure of Fig. 3.7. The no noise part is divided into two stages, with and without changes in the orientation angles.

Since we use our modified version of the SolvePnP function, we need to supply bounding information for the optimization process. This information is obtained from the simulation with or without noise depending on the simulation parameters.

3.3.3 Real-world Experiment Setup

In our experimental setup based on real hardware, we used a low cost sensor board GY86 which includes the following sensors: barometric pressure sensor [71] which was used to extract the altitude difference between points B and C in Fig. 3.2, magnetometer sensor [72], and a 3-axis

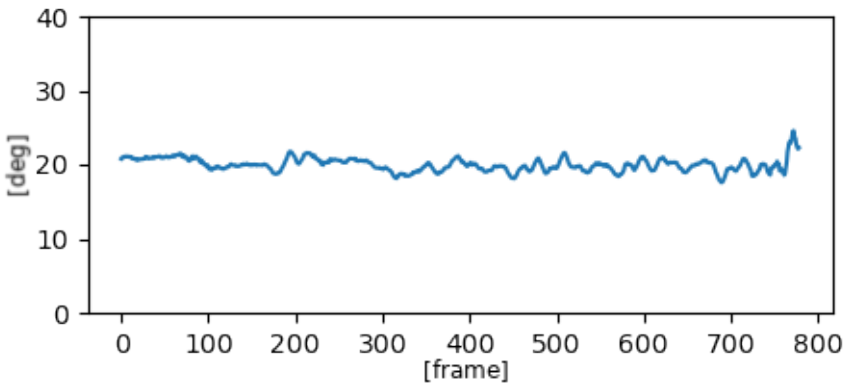


Figure 3.3: Angle between magnetic vector and g vector (from the accelerometer) is steady around 20 degrees

gyroscope with a 3-axis accelerometer [73]. We also used an arm development board [74] to communicate with the GY86 in real time. The development board was connected to a PC for data logging. We used the PS3 Sony Eye Camera [75] for image acquisition. To calibrate the camera we used the calibration toolbox from OpenCV and extracted the distortion and the camera matrix using a chessboard pattern showed in Fig. 3.4. Optitrack system [76] was used to track the camera movement during the maneuver described in Fig. 3.2. We placed 4 markers on top of the camera as shown in Fig. 3.4.

Magnetometer Calibration

We used a simple method to calibrate the magnetometer. We rotated the sensor around each axis and obtained the maximal and minimal values of the reading. Then we scaled the reading according to those measurements resulting in a calibrated magnetic vector. This was a first order calibration which did not take into account distortion but was accurate enough for our needs.

Accelerometer Calibration

We put the sensor in the direction of each axis in both direction (6 in total) and scaled the measured segment to 1g which is $-0.98m/s^2$.

Calibration and setup validation

We measured the angle between the magnetometer and the accelerometer measurement vectors and presented in Fig. 3.3. This angle should be close to constant no matter the orientation of the sensor and should be roughly 20 deg in our current world position.

Extraction of 3D Orientation

We proposed a method to evaluate camera position based on the magnetometer readings and the accelerometer readings. We used the simple TRIAD [77] method to obtain the orientation based on the two vectors: the magnetic field vector and the gravitational field vector. We validated this method by calculating the angle between those two vectors over time which should be close to constant. This is a straightforward and fast way to obtain orientation and worked well in a controlled environment and was accurate enough for bounding the least squares algorithm. For obtaining orientation in the general case higher accuracy methods such as ASGD and AHRS [78] [69] should be used. The computation of the rotation matrix was

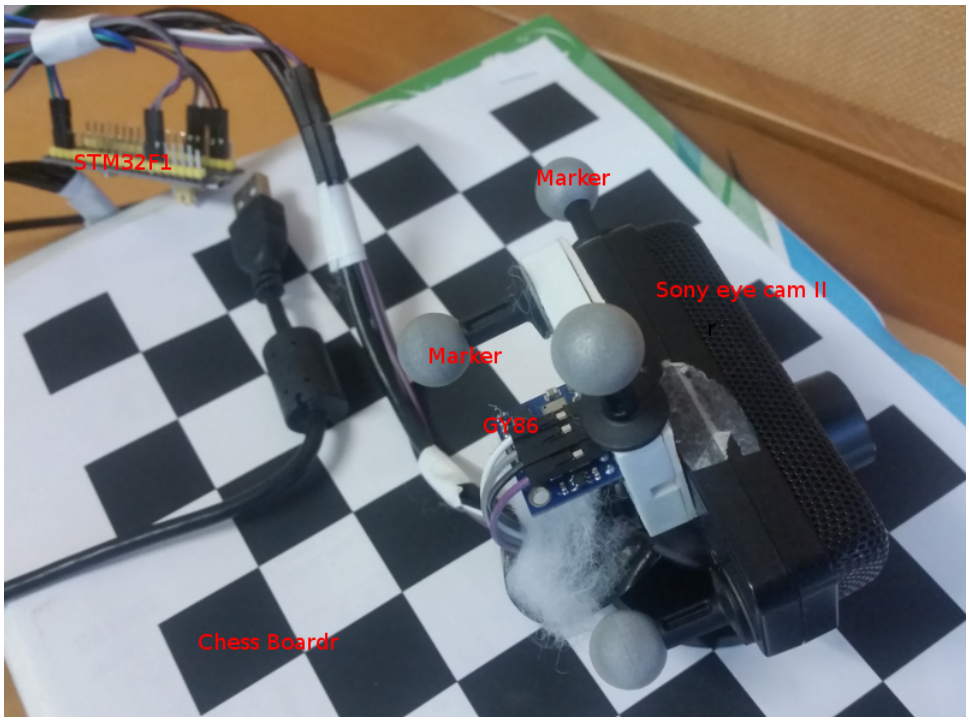


Figure 3.4: Hardware Setup - Sony PS3 eye Camera together with GY86 sensor board. Optitrack markers mounted on top of the camera and a STM32 development board used to communicate with the sensor board. The checker board used for calibrating the camera also can be seen in this image.

done as shown in the next equation:

$$R = \begin{bmatrix} Acc \times Mag \\ Acc \\ (Acc \times Mag) \times Acc \end{bmatrix} \tag{3.8}$$

where Acc is the calibrated accelerometer vector and the Mag is the calibrated magnetometer vector. We now define R_0 as can be seen in (3.7) to be the initial relative rotation (Fig. 3.2 Pt. B) and the rotations are measured relative to that rotation and R_0 is used as a reference point.

3.4 Performance Evaluation

In this section we will describe the diffrent methods and results we used to evaluate performance. We will present the results in a series of figures, and to help the reader navigate between the diffrent figures we summarise the purpose of each figure in Table 3.1:

3.4.1 Evaluation Using Simulation Setup

In this subsection we discuss the results of analysis performed using the experimental setup presented in the previous section. Fig. 3.2 depicts the path defined for camera motion through a sequence of points B to H.

In Fig. 3.6, we can see that when SolvePnP2 is not bounded the two versions of SolvePnP (1 and 2) show similar behavior in the presence of an error in the position of the 3D points (landmarks) and also similar performance when no noise was introduced to the simulation (figure 3.5). In the simulation analysis we didn't observe significant changes between SolvePnP2 and SolvePnP3 (unlike the real-world hardware setup results), so we have presented results are

Figure	Environment Type	Purpose
3.5	simulation	comparing SolvePnP1 and SolvePnP3 without added noise
3.6	simulation	comparing SolvePnP1 and SolvePnP3 (without bounding) with added noise
3.7	simulation	comparing SolvePnP1 reconstructed angles with and without added noise
3.8	simulation	SolvePnP3 Constrained version performance with added noise
3.9	simulation	Estimated angles (Euler angles) with and without noise using SolvePnP3.
3.10	real world	compares between methods SolvePnP1 and SolvePnP3.
3.11	real world	compares between methods SolvePnP1 and SolvePnP3 angles.
3.12	real world	compares between methods SolvePnP2 and SolvePnP3.
3.14	DroneSimLab	SolvePnP1 performance.
3.15	DroneSimLab	SolvePnP2 performance.
3.16	DroneSimLab	SolvePnP3 performance.

Table 3.1: Table of figures.

only from one (SolvePnP3). Fig. 3.7 shows the performance in restoring the orientation of SolvePnP1. In this figure we can see that it was unable to estimate the orientation correctly. Fig. 3.8 and Fig. 3.9 show the position and angle analysis of the SolvePnP3 constrained version. In the simulation, the benefit of constraining the SolvePnP function is obvious compared to the unconstrained version.

In terms of computation time our version of SolvPnP was 5 times slower then the OpenCV version and took approximately 15 millisecond per frame. This kind of tradeoff should be taken into account in system design stages.

3.4.2 Evaluation Using Hardware Setup

Figures 3.10, 3.11 and 3.12 show the results obtained using the hardware setup presented in section 3.3.3. We can see the benefit of using bounded methods especially in an unstable noisy environment which is often the case in computer vision problems. It is clear that SolvePnP2 and SolvePnP3 give much better performance in reconstructing the original course. In Fig. 3.12 we can see the difference between SolvePnP2 and SolvePnP3 which is due to the difference in the representation of orientation.

3.5 Simulating Sensor Fusion Experiments

The previous section explains in detail our proposed sensor fusion methodology. In this section we added the results using the DroneSimLab. In out research we performed experiments in three different environments. Two of them demonstrated in the previous section and they are: (a) synthetic camera simulation created by projecting dots on a simulated camera plane (b) real environment in a drone lab having Optitrack system [76] as a ground truth reference. we added a test in the DroneSimLab that correlated the results showed previously and are presented in

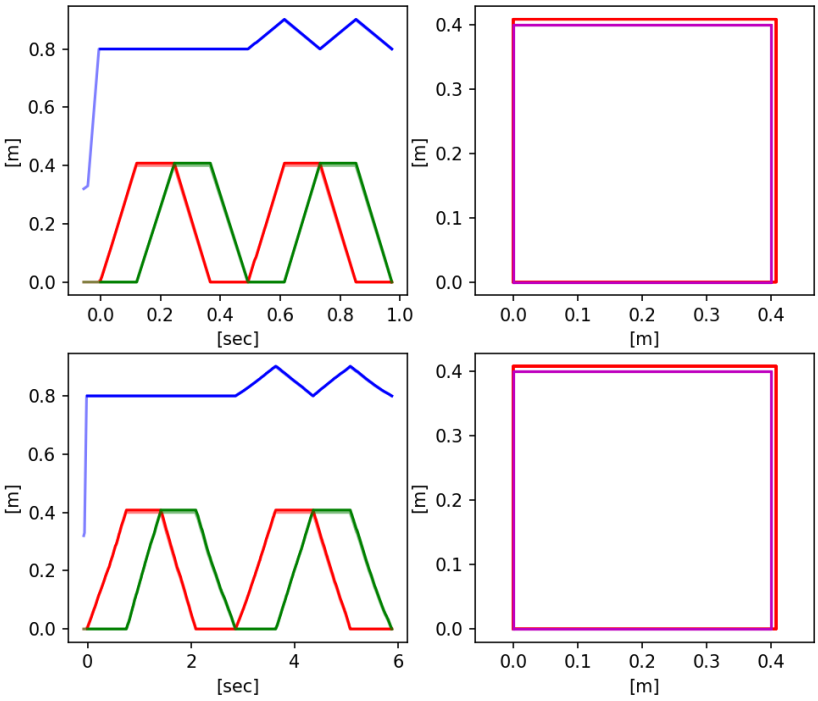


Figure 3.5: Noise Comparison - the left graphs represent camera position in 3D coordinates (meters) with respect to time (sec). The RGB color scheme corresponds to X,Y and Z 3D position where the semi-transparent colors represent ground truth, but in this case they are overridden since the ground truth is almost equal to the output of the SolvePnP1. On the right side of the graphs we can see the 2D plot of the X,Y positions (meters) of the camera where in red is the output of the methods and in magenta is the ground truth. Again the markings are overlapping. We can see that without noise SolvePnP1 and SolvePnP3 behave the same and successfully recreated the course.

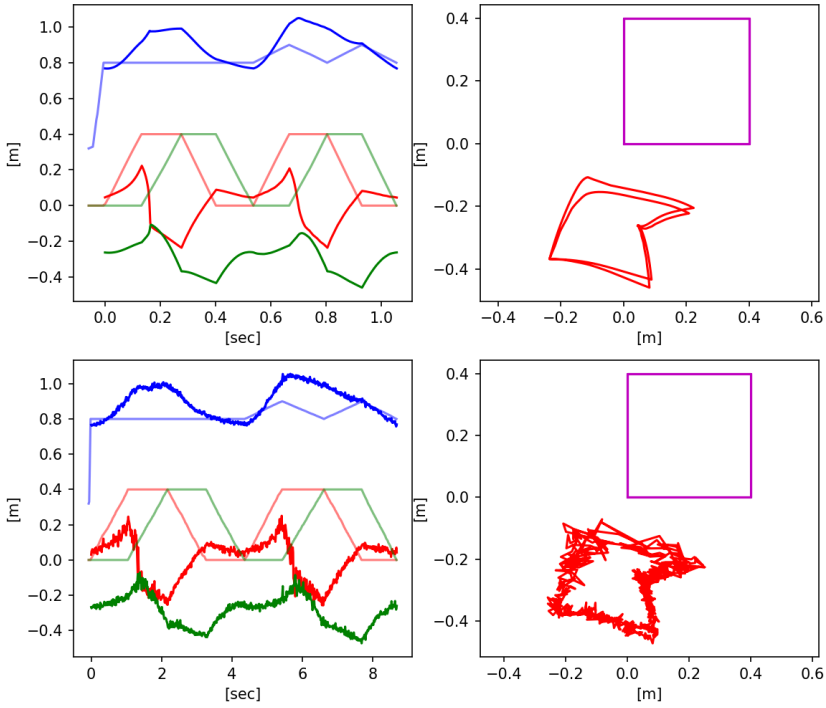


Figure 3.6: Unbounded Noise Comparison - the left graphs represent camera position in 3D coordinates with respect to time. The RGB color scheme corresponds to X,Y and Z 3D position where the semi-transparent colors represent ground truth. On the right side of the graphs we can see the 2D plot of the X,Y positions of the camera where in red is the output of the methods and in magenta is the ground truth. The comparison is between the behavior of SolvePnP1 and SolvePnP3 under noisy environment. We can see that the two different methods show similar error behavior when the SolvePnP3 is not bounded

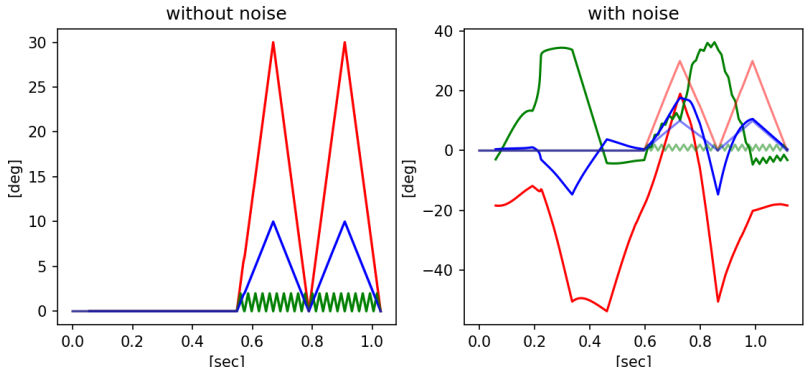


Figure 3.7: Estimated angles (Euler angles) running simulation with and without added noise using SolvePnP1, where the RGB color scheme corresponds to Euler Yaw, Pitch and Roll respectively and the semi-transparent colors represent the ground truth angles. The camera tilting was done in a triangular wave pattern between 0-30,0-10,0-3 deg for the yaw, pitch and roll respectively. When no added noise conditions introduced, the recreated saw-tooth pattern was recreated perfectly. We can see that in the noisy conditions the SolvePnP1 method was unable to reconstruct the angles.

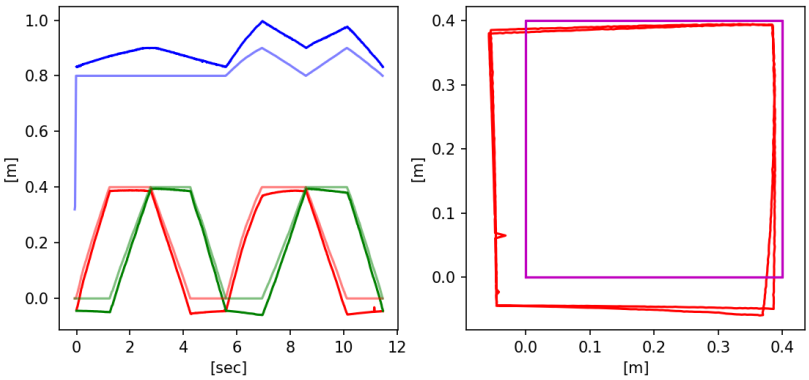


Figure 3.8: SolvePnP3 Constrained version performance. This is the same plot type as Fig. 3.5. This figure shows the advantages of the constrained version compared to the results in shown in Fig. 3.5 under the same noisy conditions

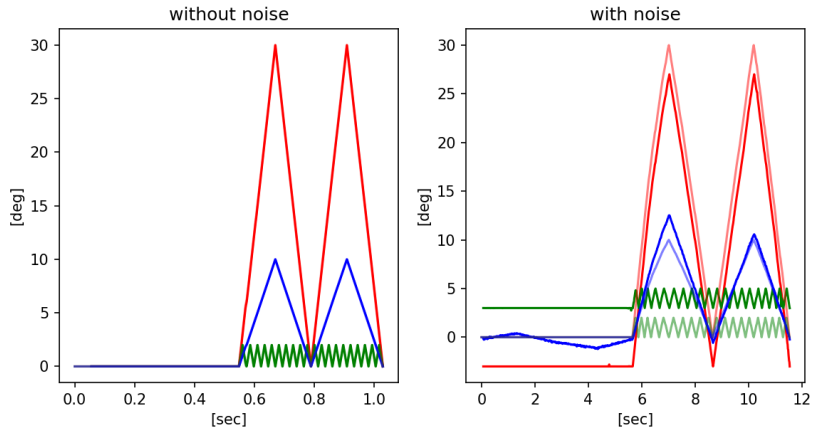


Figure 3.9: Estimated angles (Euler angles) with and without noise using SolvePnP3. This is the same figure type as Fig. 3.7. This figure shows the behavior of the SolvePnP3 under noisy conditions and the output estimated angles. from the X axis we can see that SolvePnP3 is 10 time slower when noise is introduced

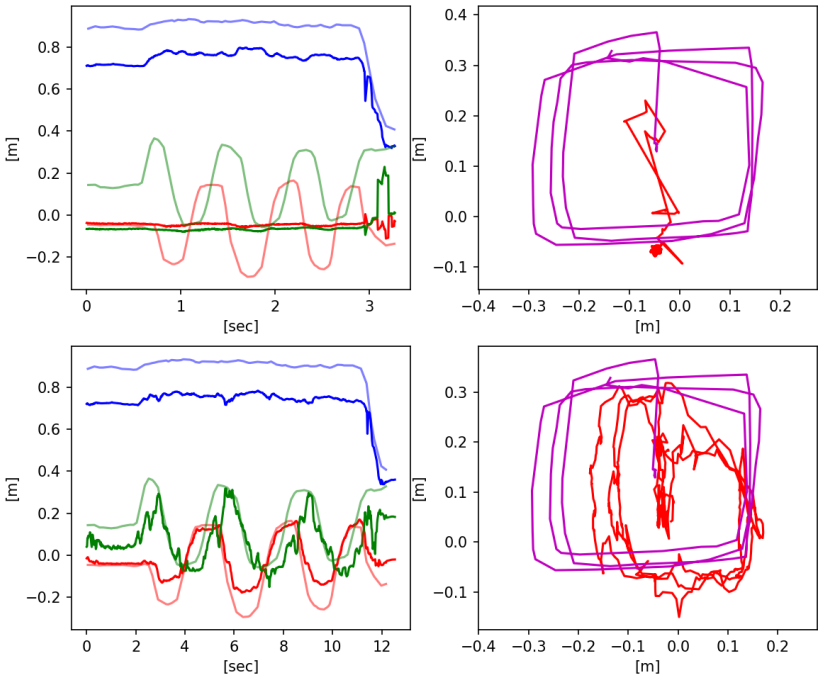


Figure 3.10: This is the same plot type as in Fig. 3.5 and it compares between methods SolvePnP1 (top row) and SolvePnP3 (bottom row) in real world experiment. We can see that SolvePnP3 is much better in reconstructing the course than SolvePnP1

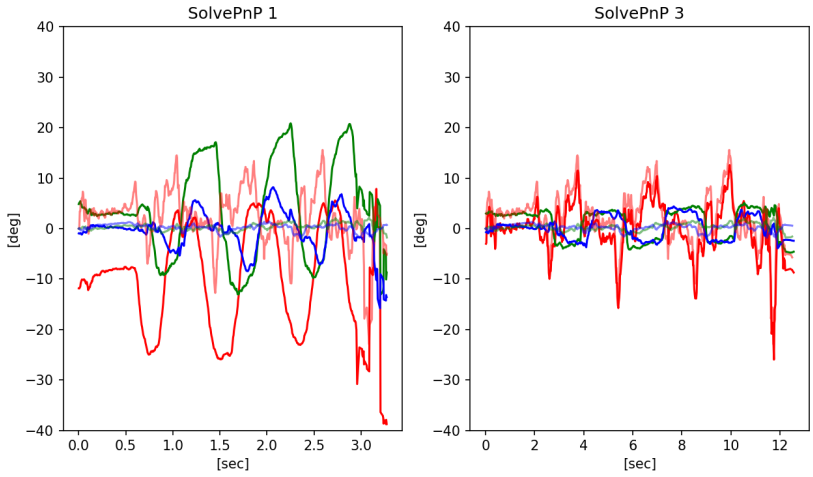


Figure 3.11: This is the same plot type as in Fig. 3.7. We can see that SolvePnP3 solution (right) is limited by the bounds whereas the SolvePnP1 (left) is enabling the solution to have undesirable angles.

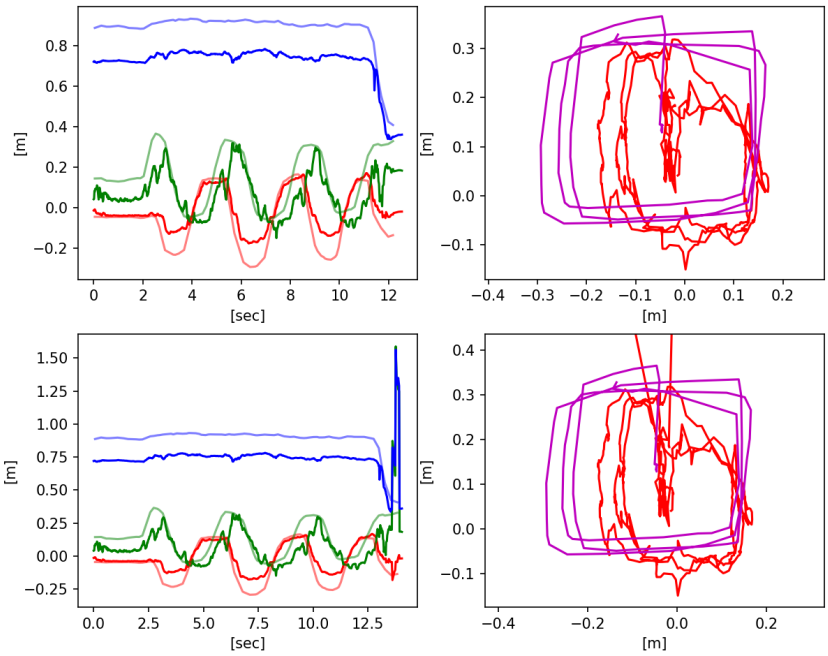


Figure 3.12: This is the same plot type as in Fig. 3.5. This figure is comparing between SolvePnP2 (top row) and SolvePnP3 (bottom row) we can see that in some conditions (at the end of the experiment) SolvePnP2 can achieve undesirable unbounded results

figures 3.13,3.14,3.15 and 3.16. These results confirmed the results demonstrated in the previous section by adding an environment similar to a natural environment to the experiments. This environment included ground truth which is difficult and sometimes impossible to achieve outside the lab. We can see that figures 3.15 and 3.16 show a significant improvement of SolvePnP2 and SolvePnP3 compares to SolvePnP1 in reconstructing the camera course. The realistic environment in that case, gave us supporting evidence to the usefulness of our approach and although we didn't include the realistic simulation in the original experiments we can see that conclusion drown from the realistic simulation were the same as those of the published research work.

3.6 Conclusion and Future Research

This chapter has presented a novel algorithm for improving the accuracy of visual odometry data by introducing bounds in the minimization of the reprojection error. The experimental results and analysis presented in this chaper show the importance of low-level sensor fusion and present a practical, low-cost implementation option for visual odometry. This emphasis of the chapter was on improving the SolvePnP function, but it combined several methods to create an understanding of position and orientation in the low-level aspect, before applying filters and other information from other sources besides inertial and tracking information. The altimeter that we used has 10 cm resolution so it can be used fully for obtaining scale information usually one order of magnitude higher than 10 cm (1 meter for example). This method can be useful in scenarios where altitude changes are frequent. The next step was triangulation from two different positions to obtain the 3D points (also known as landmarks). This method is useful when using monocular vision and can be problematic when there is a movement in the field of view which can result in significant errors in the triangulation process which make

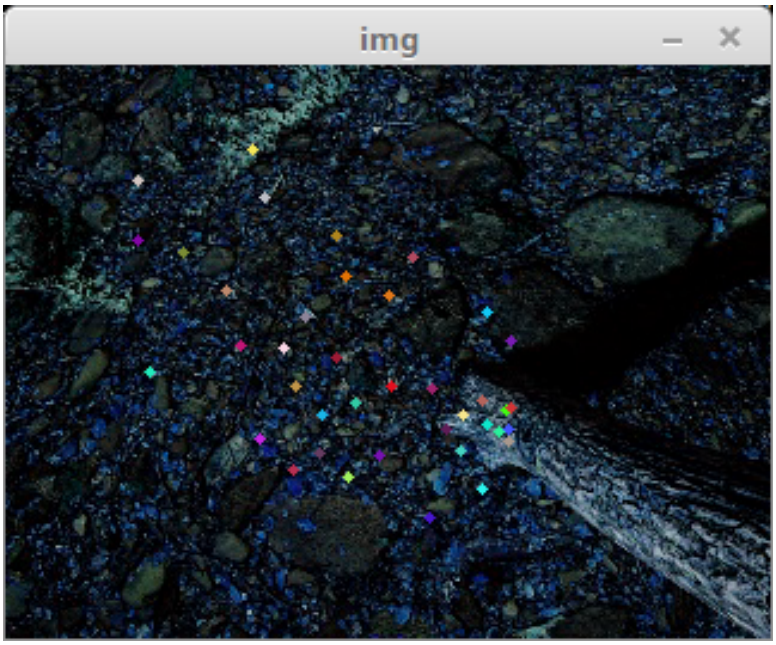


Figure 3.13: Simulated scene: performing a rectangular motion above the ground in the vicinity of a tree trunk.

the constrained methods (SolvePnP 2/3 constrained) more critical. During the rectangular maneuver, we applied our versions of SolvePnP method which showed significant improvement as discussed earlier. We can extend this current scenario implementation to the continuous case by replacing the lost features (caused by changing the field of view) with new ones and re-triangulating them with the currently obtained information if there is enough base line for the triangulation. Also, image matching techniques can be used to overcome gaps or momentary loss of tracking.

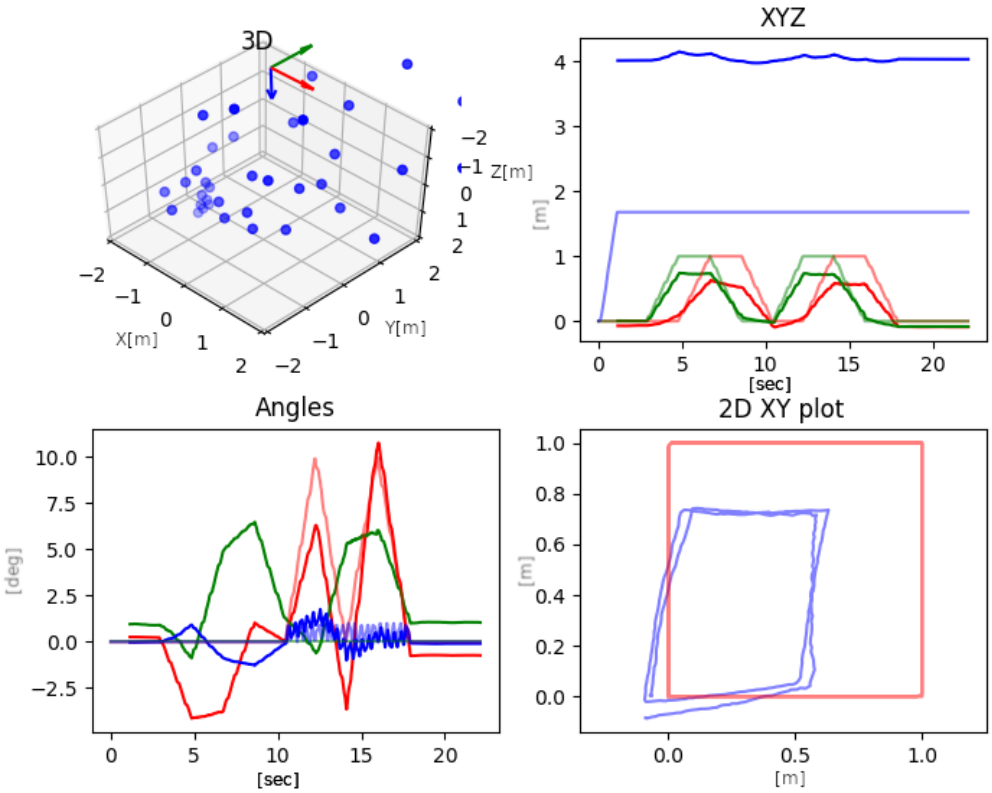


Figure 3.14: Results of SolvePnP 1 in maneuver reconstruction. The upper left image shows a 3D plot of the camera position and the extracted triangulated feature points. The upper right image shows position with respect to time of each axis of the camera where the RGB colors correspond to X,Y, and Z positions. The semi-transparent colors represent ground truth. The lower right image shows reconstructed Euler angles with respect to time where the semi-transparent lines represent ground truth and the RGB colors correspond to Yaw,Pitch and Roll. The lower right graph show 2D plot in meters showing the reconstructed course in blue vs the ground truth in red.

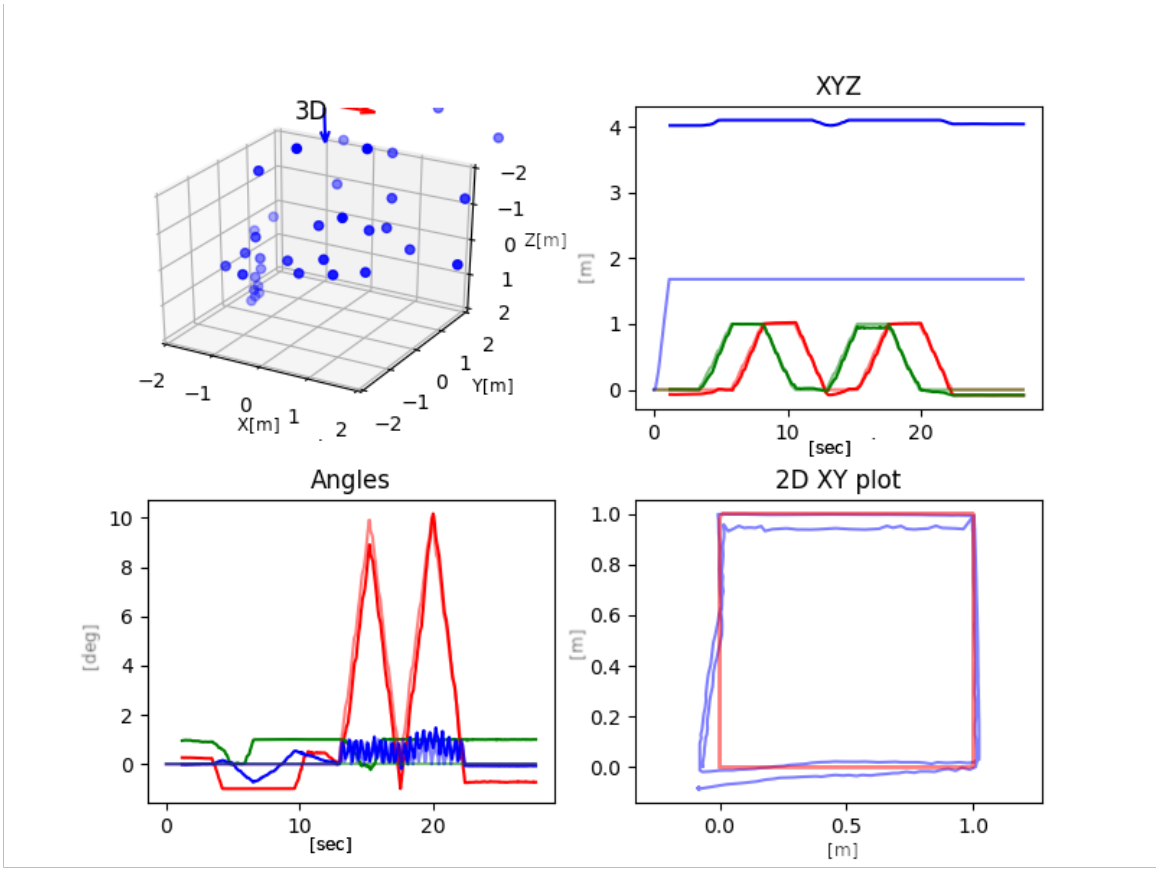


Figure 3.15: Results of SolvePnP 2 in manoeuvre reconstruction. A similar plot as figure 3.14 showing much better performance of SolvePnP 1

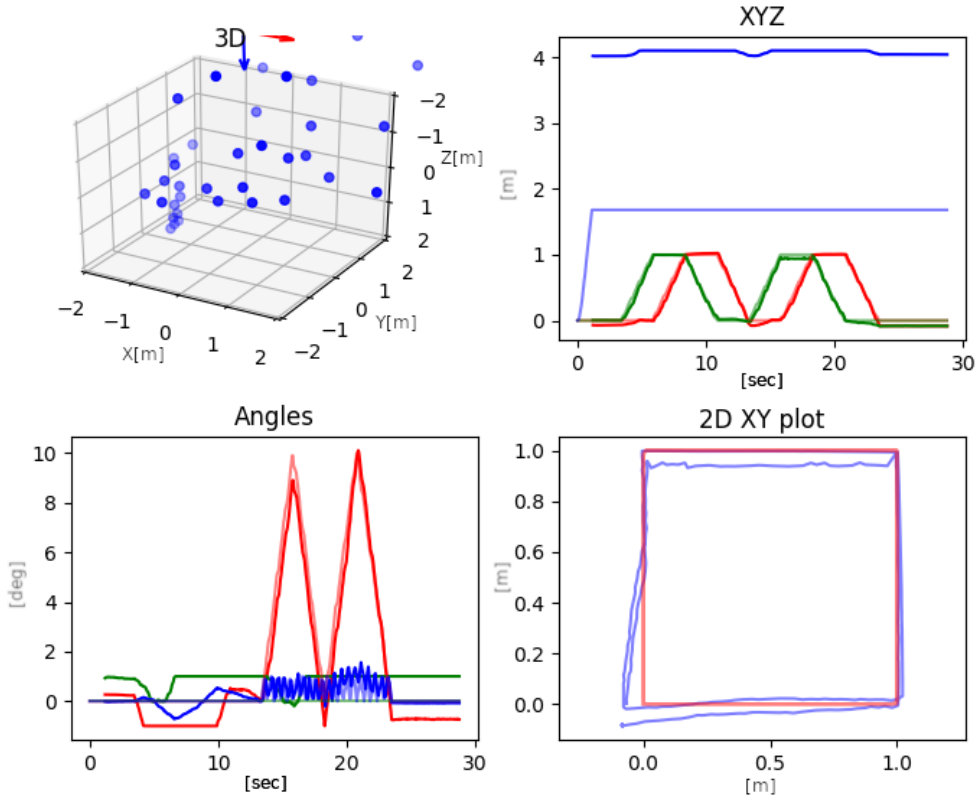


Figure 3.16: Results of SolvePnP 3 in manoeuvre reconstruction. A similar plot as figure 3.14 showing much better performance of SolvePnP 1 but very similar results as SolvePnP 2.

Chapter 4

Conclusions and Summary - Drone domain

In the drone simulation domain we focused on natural environments. Natural environments tend to encapsulate the impact of environmental conditions more then man made objects. Trees and rocks have complex surfaces and create complex shadows. When we look at a tree from different angles we see different parts of the tree compared to a building which is structurally much more well defined and each face can be projected to different angels of view. As we could see in the example of wind and feature tracking.

In addition, we showed how relatively easy it is to harness environments created specifically for gaming purposes to the scientific computer vision domain. We used existing assets to create our own virtual world and demonstrated high fidelity computer vision experiments with the ability to control the environmental conditions. In a follow up research we continued to manipulate the created environment by changing the lighting conditions as seen in figure 4.1 in order to test concepts of drone equipped with lighting for dark cases scenarios.

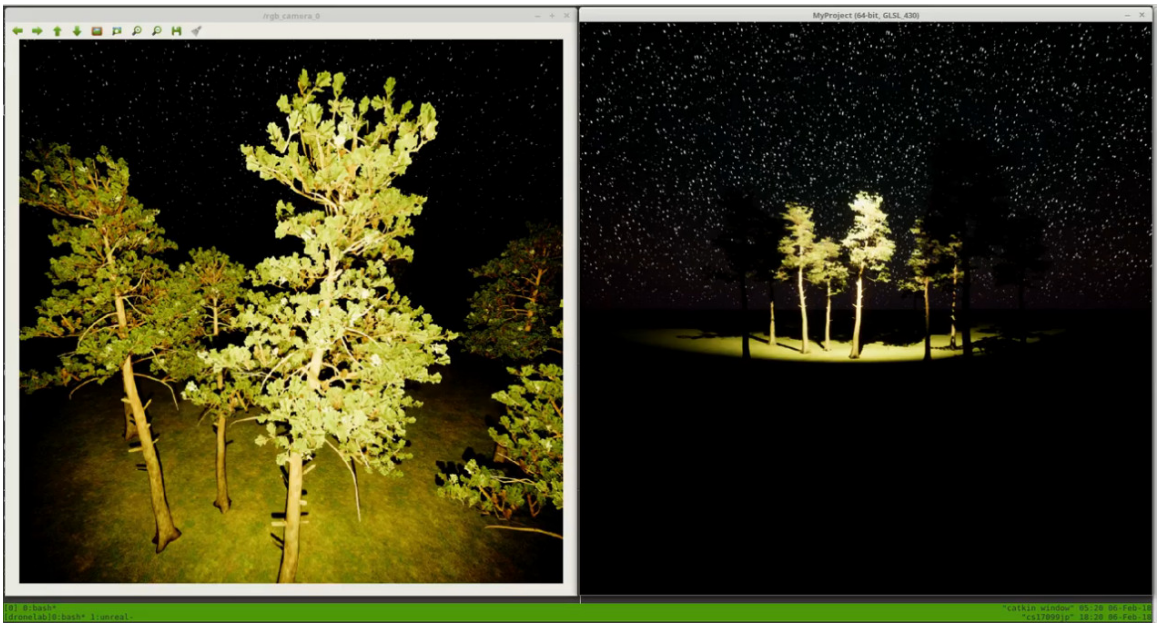


Figure 4.1: Drone at night simulation. The drone is equipped with a projector which is directed forward. The left image shows the view from the drone camera and the right image shows the view from an outside camera looking at the scene. The light projected from the drone is generating shadows directly behind the trees from the drone point of view.

In this part of the research we presented several experiments with multiple drones in a natural simulated environment. We used the software of existing drones and planted it inside

the simulation. This concept made our simulation even more realistic, not just in terms of the visual aspect but also system wide.

Up until this point we focused only on the simulation and didn't provide a practical experiment for comparison. The sensor fusion experiment was our first practical test case. This was a practical problem which we used as a probe to compare realistic simulation to real experiment. The realistic simulation experiment showed similar behaviour to the real experiment which supports our research hypothesis.

Part III

ROV simulation Domain

Overview

After exploring various underwater environments, we decided to add two missing components to game engine assets which were missing from the marketplace. The first component is an underwater tether simulation, as the existing models do not behave as expected underwater. The second component is plankton simulation which has a profound effect on the performance of the underwater vision-based system.

The next research work develops a visually realistic underwater environment for ROVs. It also includes a detailed general process for simulating the dynamics of an existing popular ROV.

The next chapter under this domain covers our experience of developing a test case algorithm for localisation in a highly dynamic and turbid environment. Our findings are based on simulated experiments and real underwater experiment.

Chapter 5

Visually Realistic Graphical Simulation of Underwater Cable

5.1 Introduction

Figure 5.1: Visual simulation of underwater tethered ROV

Cables are used widely in underwater environments for power supply and communication to remote locations and to support various types of underwater structures. High tension cables are generally used to tow fishing equipment and research probes whereas low tension cables and ropes are used in underwater tethered systems like Remotely Operated Vehicles (ROVs) (Figure 5.1). The drag introduced by the water medium and the buoyancy forces make the underwater cables behave differently to less dense mediums like air or vacuum. For most ROV cables (and also in our simulation), gravitational effects can be ignored since as a design goal, underwater ROV systems use neutrally buoyant cables. Simulating the behaviour of the cable attached to an ROV can be helpful in several situations such as (i) developing control algorithms for avoiding cable tangling, (ii) handling vision issues when the cable is in front of the ROV's camera, (iii) estimating the configuration of the cable in different conditions and manoeuvres, and (iv) simulating complex manoeuvres for missions involving multiple cables and ROVs.

The behavior of a rope under external forces is generally modelled as a multi-rigid-body dynamic system (or chain system), by representing the rope as long chain of segments. In order to simulate a large number of segments in real-time, we need a fast and memory efficient method which need not necessarily be physically accurate. At the same time, the motion should look physically realistic, and controlled by a number of parameters which defines the cable behaviour. In addition, some of the unique characteristics of the water domain need to be parameterized and added to the simulation model. Simulation of dynamic systems in computer graphics mainly use force-based methods, where linear and rotational accelerations are computed from forces and torques. A time integration method is then used to update the velocities and positions of the object. In contrast, geometry-based methods work directly on vertex positions, modifying them using iterative update equations. The main advantage of a position-based approach is its controllability. In force-based systems, overshooting problems associated with explicit integration schemes can be avoided. In addition, collision constraints can be handled easily and penetrations can be resolved completely by moving points to valid locations [79]. The position based simulations which were originally developed for the simulation of solids were also extended to the area of fluid dynamics [80] where it is not reasonable to simulate the interacting forces between the particles in real-time. In contrast to force based simulation, position based models can scale up and can be used today to simulate large multi-body systems like fabric or fur in real-time. While force based systems tend to be physically accurate under certain assumptions, position based systems aim to be visually realistic. For example, a set of simulation parameters may not correspond to real physical values, but may generate realistic object behavior as the output. The stiffness parameter implemented in the Unreal Engine which we will later discuss is an example of a parameter which influences the dynamic behaviour of the position based cable model but has no meaningful physical value (like mass or length).

Our proposed simulation method in terms of number of particles lies between the force based system and the position based system. In simulating a long cable with a large number of segments, we aim for realtime performance by applying constraints and also use position based models. In our underwater rope implementation, we take the existing Verlet integration suggested by Jackobson [81] and modify it to simulate a realistic underwater cable.

5.1.1 Our Contribution

The existing rope simulation in Unreal Engine performs in a convincing way only in a light density environment like air but looked quite non-realistic in the aquatic medium. This motivated us to return to the original assumptions of the simulated implementation and modify it in order to create visually convincing underwater rope/cable simulation. In our work, we added extensive damping and random displacements to the particles and experimentally analysed and verified the resulting behaviour. Specifically, we were interested in the behaviour of variable length long cables attached to a robot at one end and to a spool in the other end as can be seen in video [82]. That kind of cables have some unique physical characteristics that could not be addressed by the current features in existing simulation frameworks. We demonstrated the behaviour of the rope as part of a larger underwater ROV simulation using Unreal Engine 4.

This chapter is organized as follows: Section 5.2 describes the related work done on underwater cable simulation in the mechanical engineering domain and the position based simulation work done on cables in the computer graphics domain. Section 5.3 describes in detail the theory of position based methods with focus on cable simulation in addition to our proposed model. Section 5.4 describes the software we used and developed for the purpose of this work. Section 5.5 analyzes the results of the cable simulation. Section 5.6 concludes the chapter with a summary of the important concepts and results presented and also outlines future work.

5.2 Related Work

Most of the underwater cable modelling was done in the mechanical engineering domain using force based methods. Cable and chain models in general are simulated using a segment based model [83]. Buckham [84] used force based methods to simulate a cable model for use in low-tension dynamics simulation. They presented a computationally efficient and novel third-order finite element technique that provides a representation of both the bending and torsional effects and accelerates the convergence of the model at relatively large element sizes. In their paper, they managed to reduce the number of state variables defining the cubic elements of the more conventional finite element approaches . Other water cable related work reported in literature involved towed cable systems. High tension systems like towed system are quite common and a lot of work has been done on that topic. Wang investigated in his paper the parameters influencing the manoeuvre of towed cable system dynamics [85]. Lambert created a model for the dynamics and control of towed underwater vehicle systems [86]. Gonzalez created a simulation of cable pay-out and reel-in with towed fishing gears [87]. Ablow simulated the behaviour of a long cable pulled in a circular pattern [88]. Some work has been done to model the bending and the stiffness of underwater cable systems [89] [90]. Most of the simulation in the mechanical domain were designed to meet specific purpose or requirement and to serve as a guide for cable system design. For the variable length case, Prabhakar [91] developed a dynamic simulation of variable length tether in a tethered underwater vehicle system.

Position based methods are used widely in the computer graphics domain. Jackobson [81] described in detail the position based model for cable simulation. His work was the basis for the current implementation in today's game engines [92]. Our work is based on the survey paper by Bender [79] on different position based methods currently used in computer graphics.

5.3 Algorithm Overview

The Unreal game engine uses the Verlet integration method for rigid multibody simulation presented by Varlet [93]. The heart of the existing rope simulation is a particle system. each particle has two main variables: Its position x_t and its velocity v_t. The new position $x_{t+\Delta_t}$ and velocity $v_{t+\Delta_t}$ are computed by applying the rules:

$$x_{t+\Delta_t} \leftarrow x_t + v_t\Delta_t \tag{5.1}$$

$$v_{t+\Delta_t} \leftarrow v_t + a_t\Delta_t \tag{5.2}$$

where Δ_t is the time step and a_t is the acceleration. For obtaining a velocity-less representation of the above scheme, instead of storing each particle's position and velocity, we store its current position x and its previous position $x_{t-\Delta_t}$. Keeping the time step Δ_t fixed, the update rule (or integration step) is then:

$$x_{t+\Delta_t} \leftarrow 2x_t - x_{t-\Delta_t} + a_t\Delta_t^2 \tag{5.3}$$

$$x_{t-\Delta_t} \leftarrow x_t \tag{5.4}$$

$$x_t \leftarrow x_{t+\Delta_t} \tag{5.5}$$

Jackobson [81] suggested in his paper that by changing the update rule to $x_{t+\Delta_t} = 1.99x_t - 0.99x_{t-\Delta_t} + a\Delta_t^2$, a small amount of drag can also be introduced to the system. This is a useful equation that can be further modified to add large drag or damping to a system in an aquatic environment. In our implementation, we added small random displacements for creating micro current effects suitable for ocean-like environment. Those micro currents prevent the rope from looking frozen in space where there are no other forces presented to the simulation. This is

usually the case in long low tension cable characterized by tethered systems. Our proposed final model can be summarized by the following equations:

$$x_{t+\Delta_t} = x_t + (x_t - x_{t-\Delta_t})D_r + a_t\Delta_t^2 + \boldsymbol{r} \tag{5.6}$$

$$x_{t-\Delta_t} = x_t \tag{5.7}$$

$$x_t = x_{t+\Delta_t} \tag{5.8}$$

where D_r is the drag coefficient with maximal value of 1 (no drag). It is set to 0.9 to introduce a large amount of drag typical of aquatic systems and is multiplied by the velocity term $(x_t - x_{t-\Delta_t})$ When the time step Δ_t is set equal to 1 for simulation purposes. $\boldsymbol{r}$ is the added random displacements to simulate random forces generated by underwater micro-currents. $\boldsymbol{r}$ was uniformly distributed and limits were chosen to be small enough so the random behaviour will only cause long-term effect on the cable.

The next step of the rope simulation is to apply the distance constraint. This means that the distance between adjacent particles should be kept constant. This process is done iteratively by pushing the particles directly away from each other or by pulling them closer to maintain the required distance. The following pseudo-code (Figure 5.2) describes this process:

```
SolveDistanceConstraint(PosA,PosB,TargetDistance):
 Delta = PosB-PosA
 ErrorFactor=(|Delta| - TargetDistance)/
                |Delta|
 PosA += ErrorFactor/2 * Delta
 PosB -= ErrorFactor/2 * Delta

SolveConstraints():
 for iter=0 to SolverIterations
  for ParticleIndex=0 to NumOfParticles-1
   SolveDistanceConstraint(
    Particles[i],Particle[i+1],TargetDistance)
  for ParticleIndex=0 to NumOfParticles-2
   SolveDistanceConstraint(
    Particles[i],Particle[i+2],2*TargetDistance)
```

Figure 5.2: Distance constraint algorithm

This pseudo-code above shows how distant constraints are implemented in Unreal Engine 4. We can see that the "SolveConstraints" function has one outer loop which is responsible to perform iterations to enforce the constraint. More solver iterations will give more stiffness to the cable. In Figure 5.4, the length of the rope is changed when introducing a force at one of its ends and changes in the overall length is dependent on the number of constraints and iterations applied to the rope particles. The second inner loop reduces the flexibility of the rope by enforcing constraints between particles that are separated by one other particle. That specific constraint limits the ability of the rope to bend. Figure 5.3 shows the difference between the two constraints. The bending constraints were useful in smoothening out the effects of random displacements added to the underwater simulation. Finally, our final solution was implemented as a new underwater rope plugin.

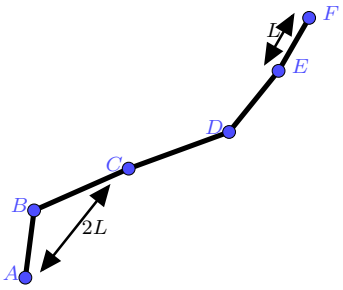


Figure 5.3: Length constraints and bending constraints. We can see in this diagram an example to the constraints apply on the cable segments. For example between adjacent points like E and F we require L distance which will create resistance to stretch and between points with one vertex between them like A and C we require $2L$ distance which will cause resistance to stretching with additional resistance to bending.

The rest of the changes to the rope characteristics were made by parameter changes to the model. Cable length was chosen to be 10 meters and the number of segments was chosen to be 100. This was done in order to create a large amount of short segments, required for visually realistic underwater simulation. With such models, any disturbance at one end of the cable will propagate slowly and will be damped by the surrounding water body.

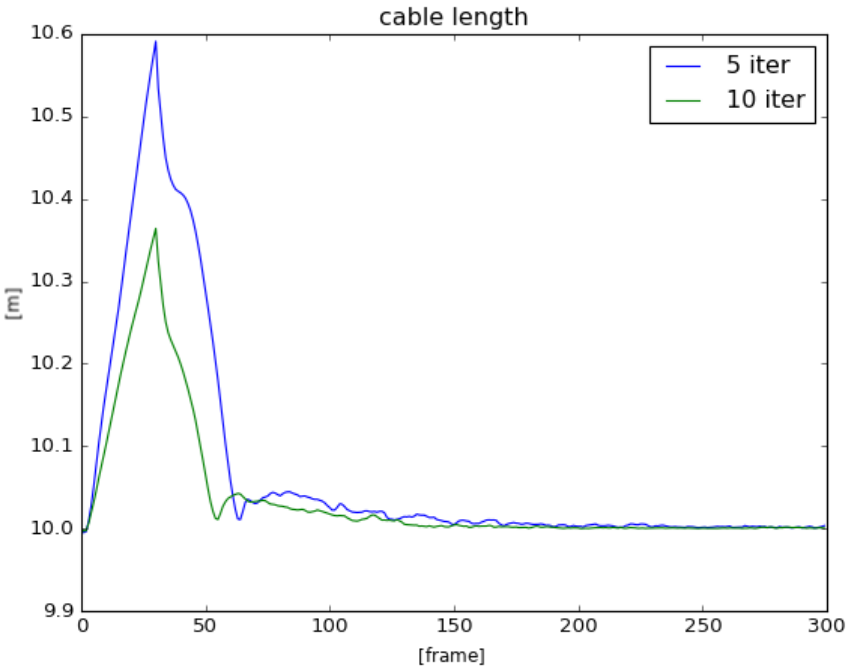


Figure 5.4: This figure shows the variation of the total cable length in meters with respect to the frame number. We can see that when using 10 solver iterations between frames to enforce the distance constraint of each cable segment the cable maintains its overall length more and represents a less stretchable cable.

The SolverIterations parameter (as can be seen in the pseudo code) should be chosen carefully. There is a trade-off between the stiffness or the ability to stretch and the damping mechanism introduced earlier. Since the damping is done in the Verlet stage, the constraint mechanism can still freely move all the rope particles, and due to that trade-off we limited the number of iterations. An improvement can be made to add some damping effect also in the constraint stage. That will allow more control on the cable length.

Setting the gravity to be zero was done to simulate the effect of neutral buoyancy. Usually, tethered systems are designed to meet the goal of neutral buoyancy to eliminate pulling forces

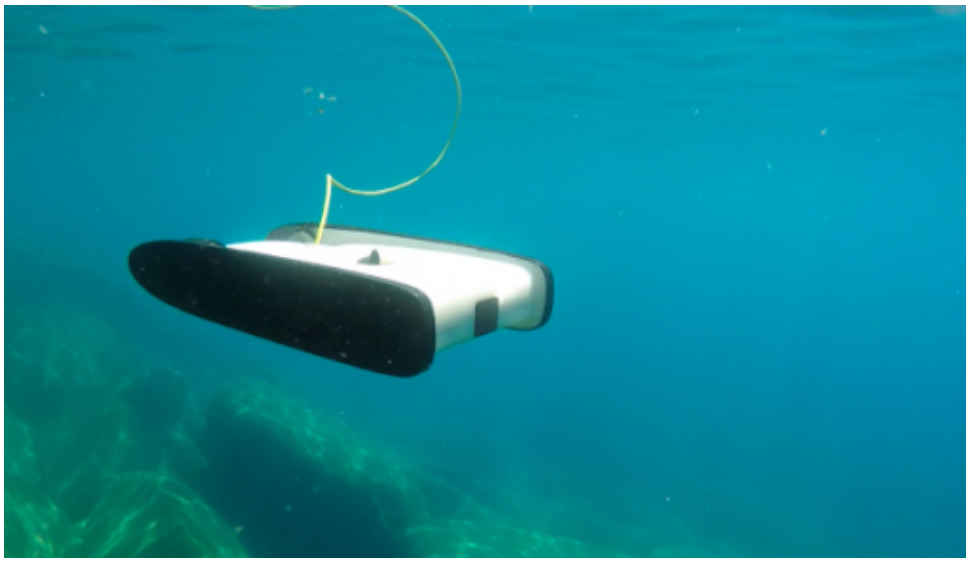


Figure 5.5: A small underwater OpenROV robot connected through a thin cable for video and control transmissions [94].

from the cable in the case of non-neutral buoyancy. Additionally, the negative buoyancy of a tethered system can cause the cable to be tangled with objects on the surface of the seafloor.

Drag coefficient was chosen to be 0.9 and this value is much lower than the maximal value 1. This was done to introduce intense damping and to reduce the propagation along the cable. The random displacements coefficient was chosen empirically to be 0.1 and can be adjusted to different sea conditions. All the parameters of the simulation included the added parameters (the damping and the random displacements) can be controlled by the outside environment (like the game engine editor) and can be adapted to different types of cables with different characteristics.

In our tethered ROV simulation, we have also made additional assumptions that there are no forces or relatively small forces introduced by the rope which effect the ROV position. In some cases, it makes sense for example if the mass of the ROV is relatively much higher then the mass of the rope. For example the OpenROV (Figure 5.5) [94] robot uses a very thin cable which handles only communication (not power) and in this case, we can assume that unless the cable is fully extended the relative force applied by the cable is relatively small. In practice, This means that the rope is not limiting the ROV movement.

5.3.1 Variable Length Cables

Underwater simulation of ROVs will also require modelling of cables connected to a spool that are released or retracted according to a naive logic that whenever there is a tension in the cable the cable is released. In the following, we outline a method to extend our model to a generate a variable length cable.

A flag is associated with each particle of the cable model, and it represents whether the particle is free to move according to the Verlet integration and the constraint mechanisms. By default, both ends of a cable will be flagged as non-free and the rest are free, since the cable is attached to both ends. In the case of a spooled cable, all the particles of the rope that are currently not released are flagged as non-free particles.

Since our model is position based, whenever the first segment from the spool side is stretched enough, typically by 10 percent of the total length, we will release a particle. After that, the Verlet integration and the constraint mechanism will move into action and will adjust the particles accordingly. In video [82] we can see that when the robot is moved the cable is pulled

as necessary to maintain low tension.

We have made further modifications in the model, particularly in the areas closer to end points. Random forces were not be applied on the first free segment, to avoid the spontaneous release of the cable due to random change of the length of the first free segment.

The spooled cable extension is done with the intention to lay a simulated foundation for the development and testing of managed tethered system. The simulation can report in real-time the current length of the cable and the estimated tension of the cable at each point along the cable. Specifically, in the beginning and the end of the cable wherein a real system we can place tension sensors as an input to the controller of the tethered system. The tension can be measured as a function of the distance between every two particles.

5.4 Methods And Tools

Figure 5.6: Unreal Engine 4 editor environment.

The main aim of this work has been to generate a convincing and realistic behaviour of an underwater tethered robot using the simulation framework provided by the Unreal Engine 4 (Figure 5.6). A live video demo can be seen in videos [95] [82] and in Figure 5.7. The experiments were done in the editor environment (not as a packed game). The robot seen in those figures was moved manually while the cable was attached to both ends. The new plugin is maintained and can be downloaded from here [46].

In addition, to have finer control over the simulation, an additional 2D simulation was created to demonstrate the proposed method. We used the Jupyter [96] python notebook environment to generate the output seen in figures 5.9 and 5.11. The 2D simulation is maintained under the following link [47]. Figure 5.8 illustrates the cable configuration used in the 2D simulation.

5.5 Experimental Results

We created a simple 2D computer simulation to simulate the effects of moving one edge of the cable while the other end is pinned (Figure 5.7). Figure 5.9 shows the behaviour of the rope

Figure 5.7: Experimental verification of motion of an extensible cable. We colored the cable in a checkers like pattern to enable the extension of the cable to be observable.

with and without damping with respect to time. The wave motion continues to propagate through the rope when there is no damping whereas with damping the wave energy slowly decays and random forces are becoming more dominant. In Figure 5.10 we can see our desired effect when using coefficient 0.9. The movement of the cable at one end does not affect the other end, so long as there is low tension in the cable segments (opposed to high tension scenarios like towing).

Figure 5.11 shows the the random forces effect. In this experiment we repeated the manoeuvre described in figure 5.8 and we look at the cable configuration in the 2D space after the system is stabilized ($t >> 0$) with and without random forces. We can see that the random forces create a kind of memory loss effect of the shape of the cable. This effect is important when there are no other significant forces (or they are close to zero) in the system. When we don't add the random force, the cable tends to stand still in contrast to what would be expected in a dynamic aquatic environment.

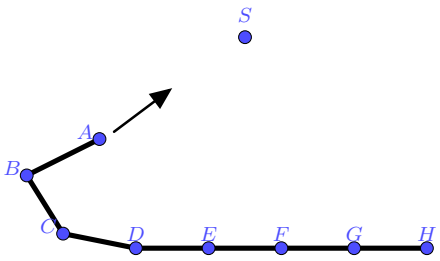


Figure 5.8: A 2D model of a flexible cable where one of the end points A is moved with a constant velocity v towards a target S

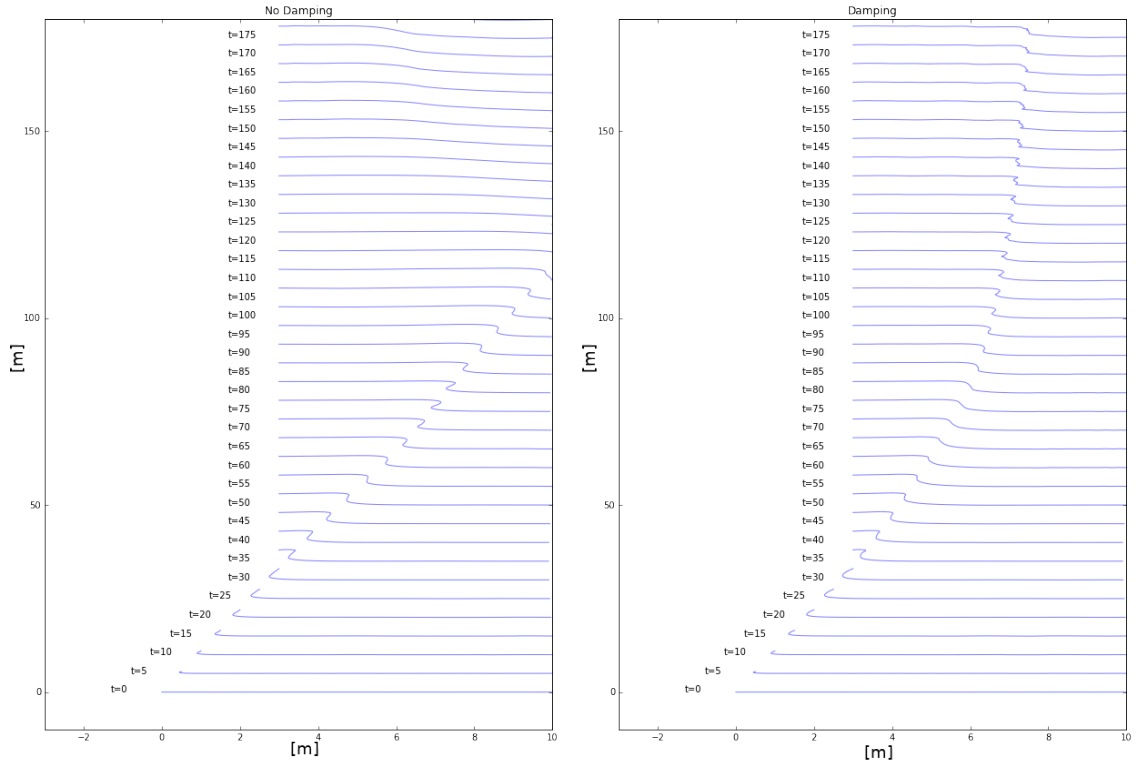


Figure 5.9: The damping effect. This figure shows the cable in different time steps. In t=0 we start to move the edge of the cable in the direction up and right along the "xy" plane. The first figure shows the results without damping and the second shows the behaviour of that cable with damping coefficient of 0.98. We can clearly see that the damping is absorbing the wave energy as we might expect in aquatic systems.

5.6 Conclusions and Further Research

In terms of performance, the modifications to the current model didn't require more computational effort. In fact, if we assume neutral buoyancy of the cable we can remove the gravitational forces from the simulation to reduce computational time. This can be useful in cases where a large number of cable/rope like object are simulated. Generally speaking, we can say that a cable is a linear 3d object curve which can be efficiently computed by modern CPUs.

Using the position based approach allowed us to easily modify the current model by adding drag and random displacements and in the future to apply other constraints for inter-rope tangling and ROV interactions. Further research will also deal with the forces applied to and by the cable to the objects that it is connected to. This can be done by measuring the length of the segments as described in section 5.3.1. Currently, we assume that there are no forces and torques applied by the rope which effect the ROV movement, and so we can improve future simulation by adding those forces to the simulation. Additionally, we added simple random displacements with even normal distribution for all the segments. In real environments that is usually not the case and the currents are influencing the cable differently for each segment. Underwater currents are more similar to air turbulence and do not contain high frequency changes.

In our research, we found this simulation to be particularly useful in cases were the robots sees its own cable as presented in Figure 5.12. This fact may disrupt computer vision algorithms - especially those based on tracking using landmark features from the images and assume that these landmarks are not moving in the scene. With this new type of simulation, that kind of behaviour can be simulated in a manner close to real cable behaviour. New computer vision algorithms can be developed to mitigate that behaviour and new control algorithms can be developed to manage the cable configuration.

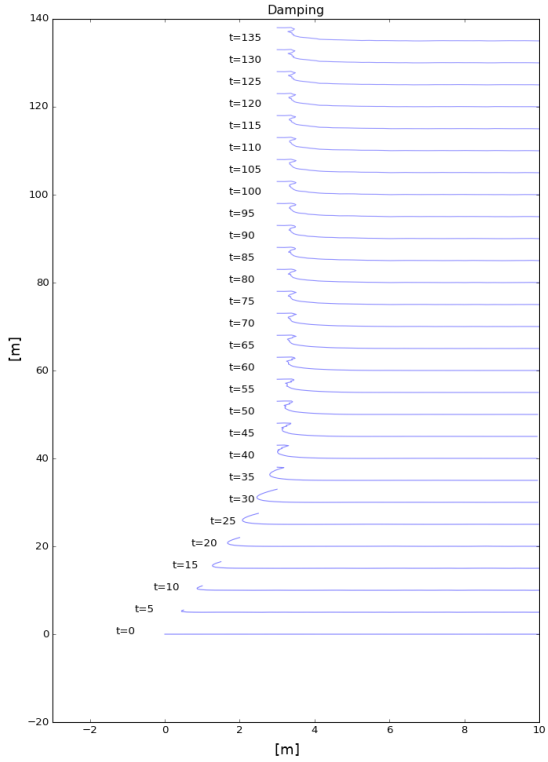


Figure 5.10: Damping with 0.9 coefficient. Setting the drag coefficient to 0.9 causes the desired effect for underwater simulation in the case of a low tension cable in an underwater environment. Disturbance on one side remains local.

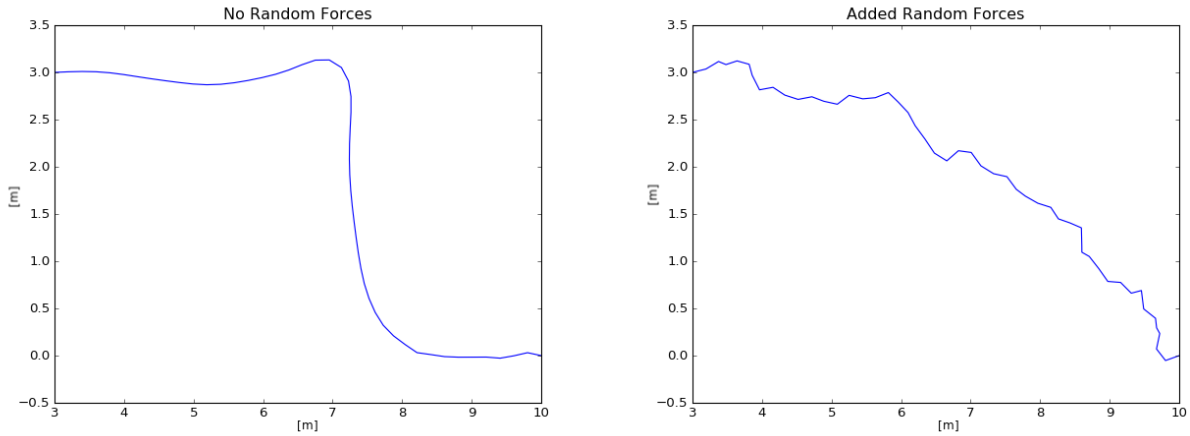


Figure 5.11: Introducing random forces to the system. Both images show the cable state after the system is stabilized ($t >> 0$). The first and the second images show the cable state with and without random forces respectively. We can see the "Memory Loss" that we would expect to see in a marine-like environment with underwater currents.

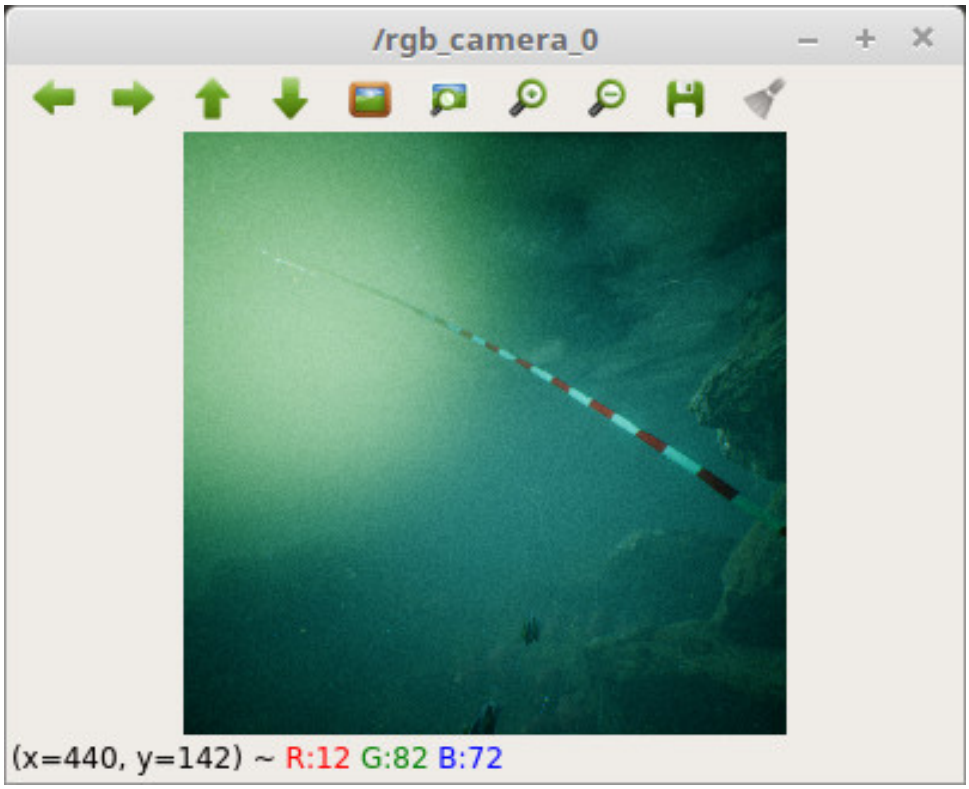


Figure 5.12: An underwater robot's camera view of its own tether cable

In this chaper, we demonstrated a visually convincing underwater cable simulation. The current state of the art model implemented in the latest version of the Unreal Engine 4 was thoroughly investigated and the needed modifications to the model for underwater simulation were described in detail. We presented a novel approach to the underwater simulation and the unique characteristics of such a medium. We showed results in a 2D computer simulation for finer analysis of the simulation results. The simulation is robust and controlled by a large number of parameters as previously described. Finally, the modified model was implemented in Unreal engine 4 as a new underwater cable component available for download with provided demos demonstrating the new cable behaviour [46].

Chapter 6

Visually Realistic Plankton Models for Simulating Underwater Environments

6.1 Introduction

Simulating visually realistic marine environment can be very challenging due to the presence of several factors affecting the illumination and motion of objects within a region. In contrast to the air medium, the underwater marine environment appears significantly more turbid and dynamic, being rich with organisms. Each organism has its own special visual characteristics. They can be for example, semi-transparent like jellyfish or completely opaque. The scene can be highly dynamic where schools of fish pass very close to the camera. Such complex and dynamic environments need to be modelled for simulating the motion of an underwater remotely operated vehicle (ROV). With increasing applications in underwater exploration, ROVs attached with RGB cameras have become more prevalent. The design of ROVs with attached computer vision systems call for the development of visually realistic simulation models where vision, dynamics and control aspects of the systems can be tested. This chapter deals with such a realistic simulation for an underwater ROV attached with an RGB camera (Figure 5.1). Specifically, the chapter focuses on the modelling aspects of bioluminescent plankton in the marine environment and the associated visual effects. Figure 6.1 presents an image we took underwater in an area rich with organic matter. The predominant fog and snow like particle effects are created by plankton. This figure clearly shows the need for realistic plankton simulation both for testing computer vision algorithms and also for human training purposes. To the authors knowledge, no prior research work has been reported on graphical modelling and simulation of plankton.

This chapter is organised as follows. The next section gives some background information regarding relevant marine environment characteristics, especially the two types of plankton (zooplankton and phytoplankton). In sections 6.3 and 6.4 we will discuss the details of simulating these two types of planktons. Section 6.5 will summarise our results and will present our followup future research.

6.2 Plankton

In this section, we provide a brief introduction to the important characteristics of a marine environment that are relevant to the development of a visually realistic simulation model.

Figure 6.1: We took this underwater image during daylight in Pelorus Sound New Zealand to demonstrate the challenges of modeling realistic underwater imagery. The image shows significant amount of organic material and reflections from the plankton. We can also observe the mist effect created by the phytoplankton

6.2.1 Marine Organisms

The marine environment supports two basic types of marine organisms. One type comprises of planktons or those organisms whose powers of locomotion are such that they are incapable of making their way against a current and thus are passively transported by currents in the sea. The word plankton comes from the Greek planktons, meaning that which is passively drifting or wandering. Depending upon whether a planktonic organism is a plant or animal, a distinction is made between phytoplankton and zooplankton. Although many planktonic species are of microscopic dimensions, the term is not synonymous with small size as some of the zooplankton include jellyfish of several meters in diameter. Nor are all plankton completely passive; most, including many of the phytoplankton, are capable of swimming. The remaining inhabitants of the pelagic environment form the nekton. These are free-swimming animals that, in contrast to plankton, are strong enough to swim against currents and are therefore independent of water movements. The category of nekton includes fish, squid, and marine mammals [97].

Plankton comprises unicellular plants "phytoplankton" or generally small (millimetre or less) animals "zooplankton" that are adrift on the currents. Phytoplankton is responsible for about 45% of global annual primary production and is grazed by zooplankton, which in turn are suitably sized food items for predators including commercially important fish and great whales. Plankton is vital component of marine and freshwater water-column ecosystems [98] [99]. The movement of plankton is dependent on ocean currents and currents created by moving object like fish movement or even robotic arm movement can shift the water from side to side which in turn will create a visible movement of plankton.

6.2.2 Seawater Optical Properties

From an optical imaging point of view, seawater is an absorbing and scattering medium. Light energy that propagates in water is absorbed by water molecules and dissolved organic matter (DOM) or suspended sediments [100]. Propagating light is also elastically scattered by thermal fluctuations in water (Rayleigh scattering) and by hydrosol particles suspended in water. The major portion of absorbed energy is transformed into heat. The rest of the absorbed energy is reemitted as Raman scattering and fluorescence. Elastic scattering occurs without a change in energy only the direction of propagation changes [101].

6.2.3 Principles of Underwater Imaging

Underwater optical imaging systems can be broadly classified into two areas: passive and active. Passive systems utilize light in order to image objects that have been illuminated by some source other than that associated with the imaging system. Examples are imaging of objects using sunlight or other sources of illumination such as bioluminescence. Active systems take advantage of a user generated source of light. A simple example is an underwater camera system which uses either strobe or continuous artificial illumination [102].

In both systems (active and passive) backscattering or diffuse reflection (reflection of light waves back to the direction from which they came) is another effect that is commonly seen as shown in Figures 6.1 and 6.3. In the active system, we can control the intensity of backscatter by reducing the intensity of the light or changing the position of the strobe relative to the camera, whereas in a passive system we need to change the camera orientation or use special filters.

Different wavelength propagates differently in water. The ratio between the absorption coefficient of the red spectrum (approx. 650nm) and the blue spectrum (approx. 450nm) is almost 40 (Figure 6.2), which means that the red part of the spectrum will be attenuated rapidly and after a short distance it would be difficult to determine whether the white light comes from the sun or from a strobe [103] [104].

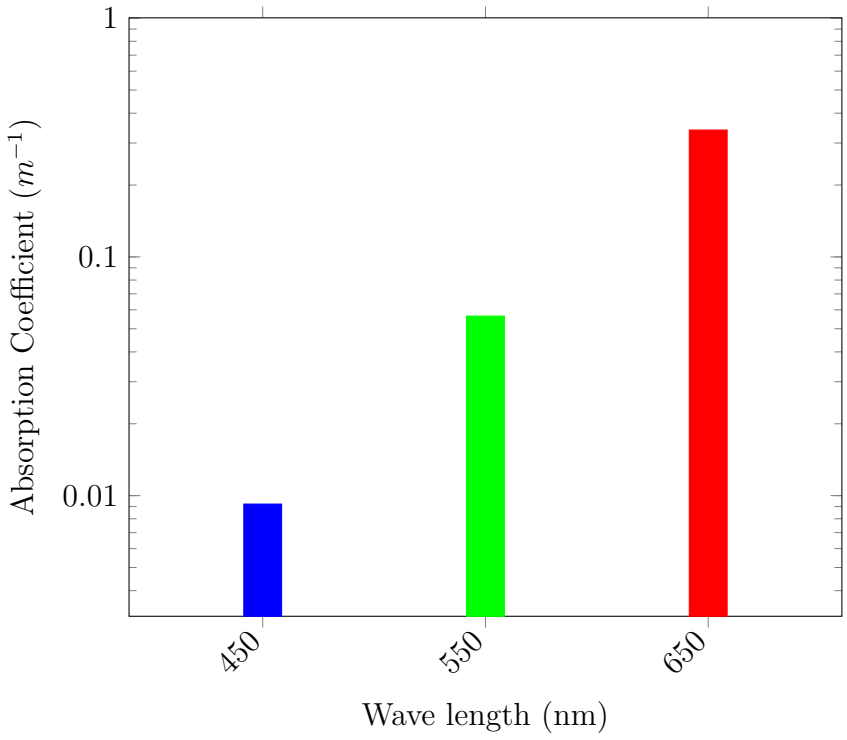


Figure 6.2: Absorption coefficient vs wavelength. Logarithmic plot showing the absorption coefficient of fresh water [103] [104].

The camera hardware plays an important part in the effects we see. Motion blur caused by camera movement is common in underwater systems. The long exposure times that are used to compensate for the lack of underwater lighting is responsible for that effect.

There are some additional optical and underwater imagery effects like the chromatic aberration [105] and Snell's circle [106] which are caused by a relatively wider field of view. These effects are not discussed in this chapter. That said, those effects which are supported by modern game engines can be easily added to the simulation [107] [108].

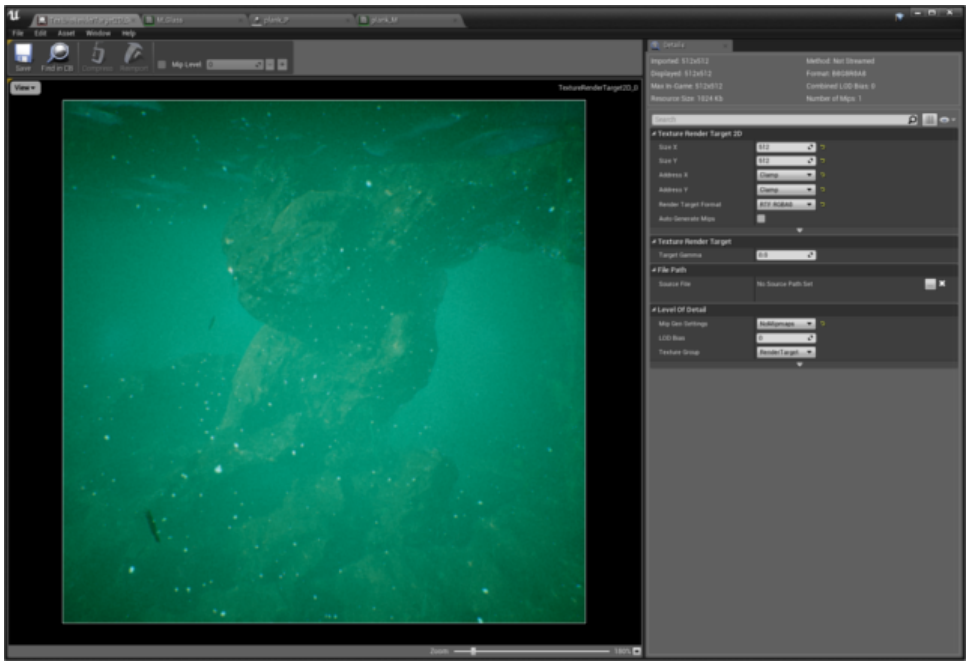


Figure 6.3: Simulating plankton effect by using particle engine in Unreal Engine 4. In this image, we can see the backscatter effect due to the active lighting coming from strobe near the camera making the snowflake like zooplankton look brighter.

6.3 Phytoplankton Simulation

Phytoplankton are microscopic photosynthesizing organisms. Therefore they are not visible directly to the naked eye. Due to the chlorophyll content in their cells, the green pigment is sometimes noticeable. We can find them in the upper layer of a large body of water exposed to sunlight. Interestingly enough, their optical properties can be simulated using Exponential Height Fog [109]. Exponential Height Fog creates more fog density in low places of a scene and less density in other places. Exponential Height Fog provides two fog colours which are used to differentiate upper and lower layers of the atmosphere. We can use this effect to our advantage by choosing a green colour for the surface level of the sea and the blue colour to deeper layers. The transition between the colours is smooth. Figure 6.4 shows the final result of simulating only Phytoplankton using the Fog component in Unreal Engine. We present two images for comparison - one is a simulated output and the other is a real image taken underwater.

6.4 Zooplankton Simulation

To simulate zooplankton which are larger and more visible particles, we used the Unreal Engine 4 particle engine [111]. In general, particle systems or engines consist of several important parts. First is the emitter which is responsible for releasing the particles. We can also control the shape of the emitter and the direction the particles are distributed. Secondly, particle dynamics specify the way the particles move through space. The particles may be randomly distributed, move along a curvilinear path, move under the influence of forces or combinations of the above. Thirdly, physical characteristics of the particles define the shapes of the particles which can be changed over time, the texture used to describe the particles, their static mesh and so on. In addition, we can also control the time of life of the particles which can be used for example to keep the system stable around a specific amount of particles.

To simulate the plankton's dynamic behaviour, we chose a flat box shape emitter which will be located close to the surface of the water. The emitter dimension was chosen to be 100x100x1

Figure 6.4: Simulating phytoplankton mist. The upper image is a real image taken underwater in Florida's east side [110]. The bottom image was simulated using UE4's "Exponential Height Fog" component. In both images, we can see the distance attenuation effect of the fog where distant objects are less visible. Both images used the sunlight as the main source of light.

meters. The XY dimension represents our region of interest due to obvious reasons it cannot be set to infinity. The Z dimension represents the depth (negative values means going deeper) and the extent where there is enough sunlight to enable photosynthes. The initial velocity is chosen to be uniformly distributed where $V_x \in U(-1, 1), V_y \in U(-1, 1), V_z \in U(1, -10)$. Under these conditions, the plankton will move randomly around the surface of the water with some preference to move to deeper layers. In that case, if the camera is not pointed directly to the emitter, the particles will slowly move in to the field of view of the camera. We assume that there are only small microcurrents that affect the movement of the plankton. This is obviously not always the case but we can later add more emitters that can simulate different underwater currents. The distribution of the particles can be seen in 2D projection in Figure 6.5 and in the 3D projection given in Figure 6.6. In those images, we can see that the initial velocity setup gives smooth transition in terms of particle density as we go deeper. This behaviour imitates the natural preference of plankton to the light coming from the surface.

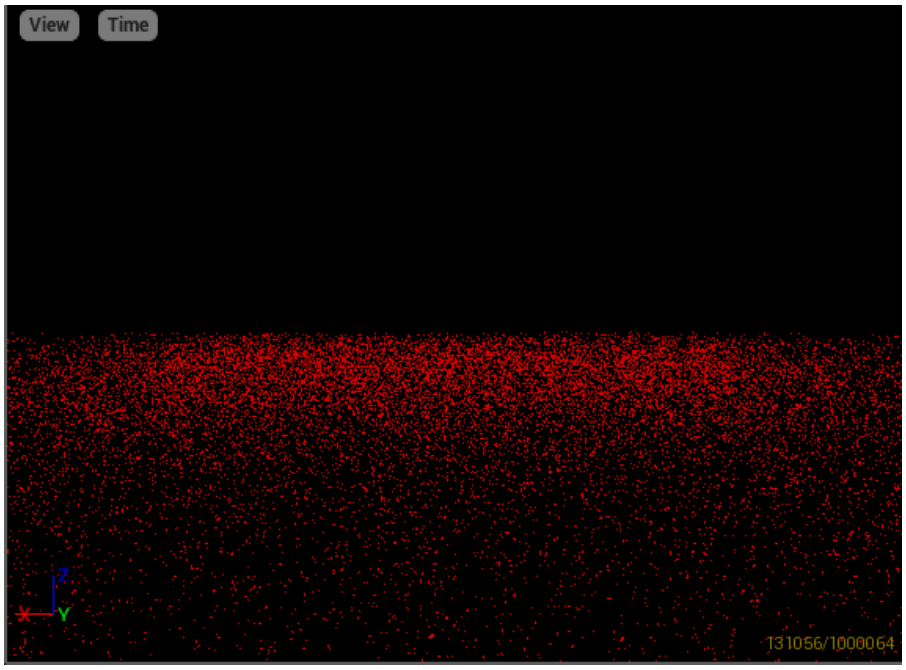


Figure 6.5: Simulated Zooplankton distribution in the particle engine. In this figure, we plot only the position of the particles in the XZ plane.

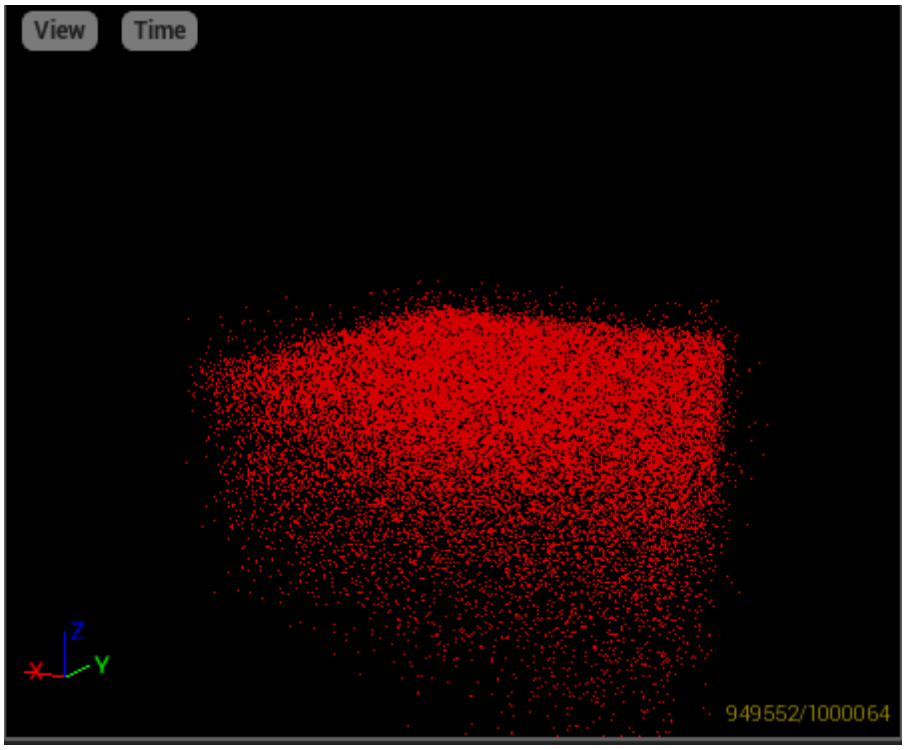


Figure 6.6: This is a 3D view of the particle emitter setup as a flat rectangular box with initial velocities pointing to the negative Z axis (vertically downward) direction.

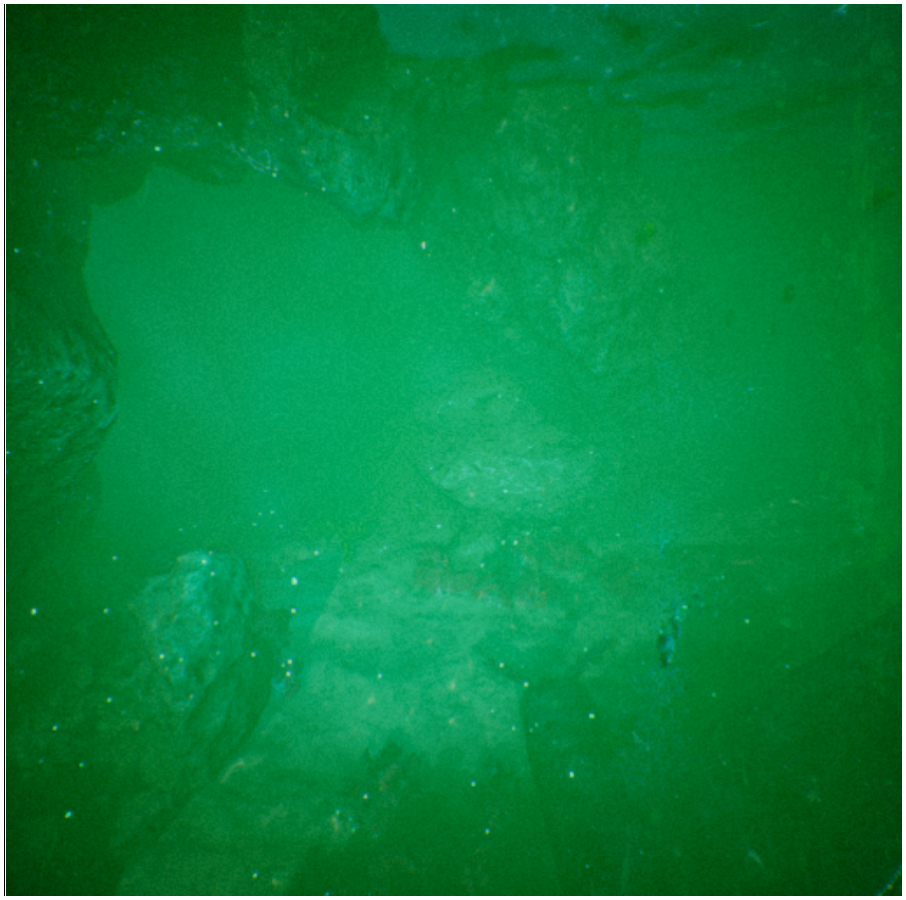


Figure 6.7: Simulated Zooplankton of different sizes and shape. The zooplankton have the appearance of snowflakes with some backscatter illumination. The scene uses a strobe light and sunlight.

In order to control the number of particles, we choose a uniformly distributed lifetime

between 80 to 500 seconds. The tradeoff, in this case, is that since the particles are constantly emitted, we need to control the number of live particles and this is done by the LifeTime parameter. This is obviously different behaviour than a real plankton particle which usually lives much longer than 80 seconds. However, due to dynamically changing lighting coming from the refraction on the surface of the water, we do occasionally see some sparkling effect which can be simulated with the addition and removal of particles as described above. In terms of visibility, plankton particles constantly move, but in a dynamic scene where the camera is also moving that should not be too significant. In contrast, although having shorter lifetime values will let the particle system stabilize faster it will come with the cost of increasing sparkling effect due to shorter lifetime of the particles and will require more particles to be emitted to preserve the same amount of living particles as with a longer lifetime value.

Another experimental approach that was tested was to use box shape emitter (not flat) (Figure 6.8) with changing size during the particle's life. In this approach, the particles don't have to move and they can appear in the scene as small particles that grow over time. We created a "size by life" in a way that the particle will grow in its first period of life and then will reduce its size slowly until it disappears (Figure 6.8). We found this approach more realistic since we have more freedom in choosing the velocity of the particles and can even set the velocity to zero which sometimes better represents some of the underwater scenes.

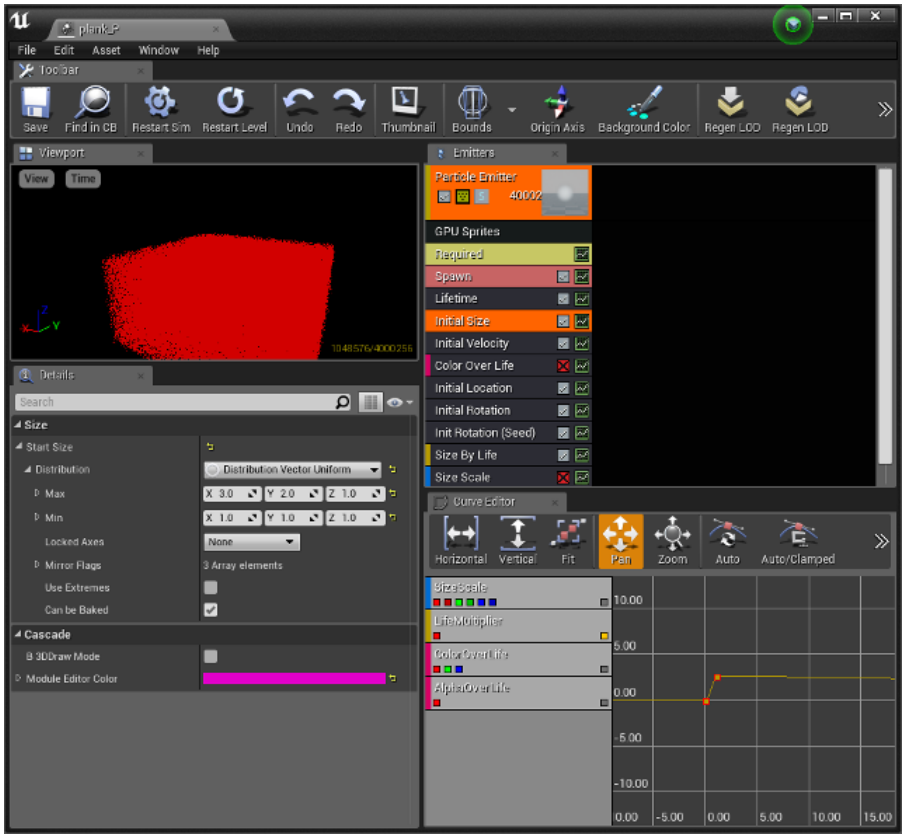


Figure 6.8: Particle engine's box shape emitter setup. In the upper left corner a wireframe plot of the box shape distribution of the particles is shown. In the right lower corner we can see the curve editor used to control the emitter parameters over time like spawning rate and particle size.

A more static approach can also be implemented by creating all the particles at once with randomly distributed size, orientation and initial velocity. In this approach, we would need to make sure that the number of particles is not slowly reduced in the case they have initial velocity. We can easily fix that issue by adjusting the spawn curve 6.8) to have initial high value in the beginning of the simulation and then small spawning rate to compensate for the particles that drifted away from our region of interest. In addition, a long time of life is also important

to keep the system stable around a fixed amount of living particles. The big advantage of that approach is that the simulation starts in almost stable state and we don't need to wait for it to stabilize. The disadvantage of the static approaches is that they create uniformly distributed particles in terms of depth. We can use different or mixed approaches with single or multiple emitters for different conditions and types of simulations.

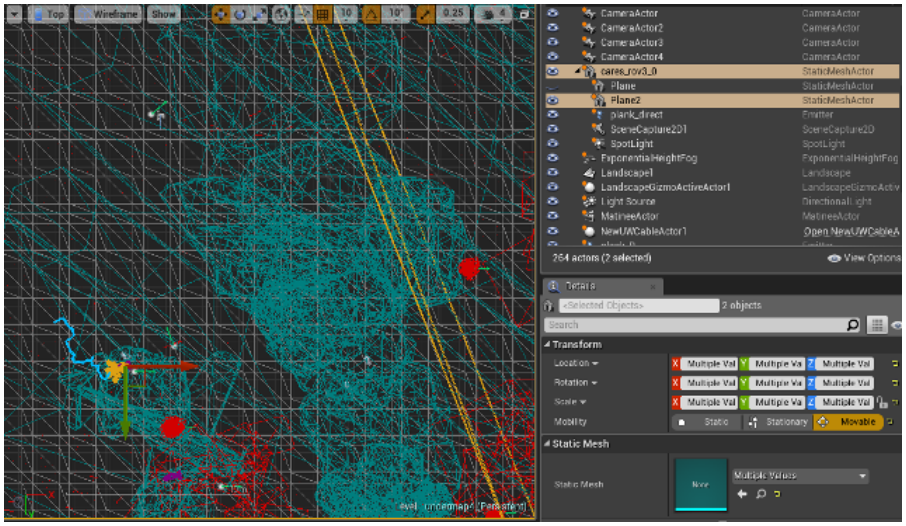


Figure 6.9: Plane and ROV top view setup. The plane in yellow was positioned in front of the ROV camera (also in yellow).

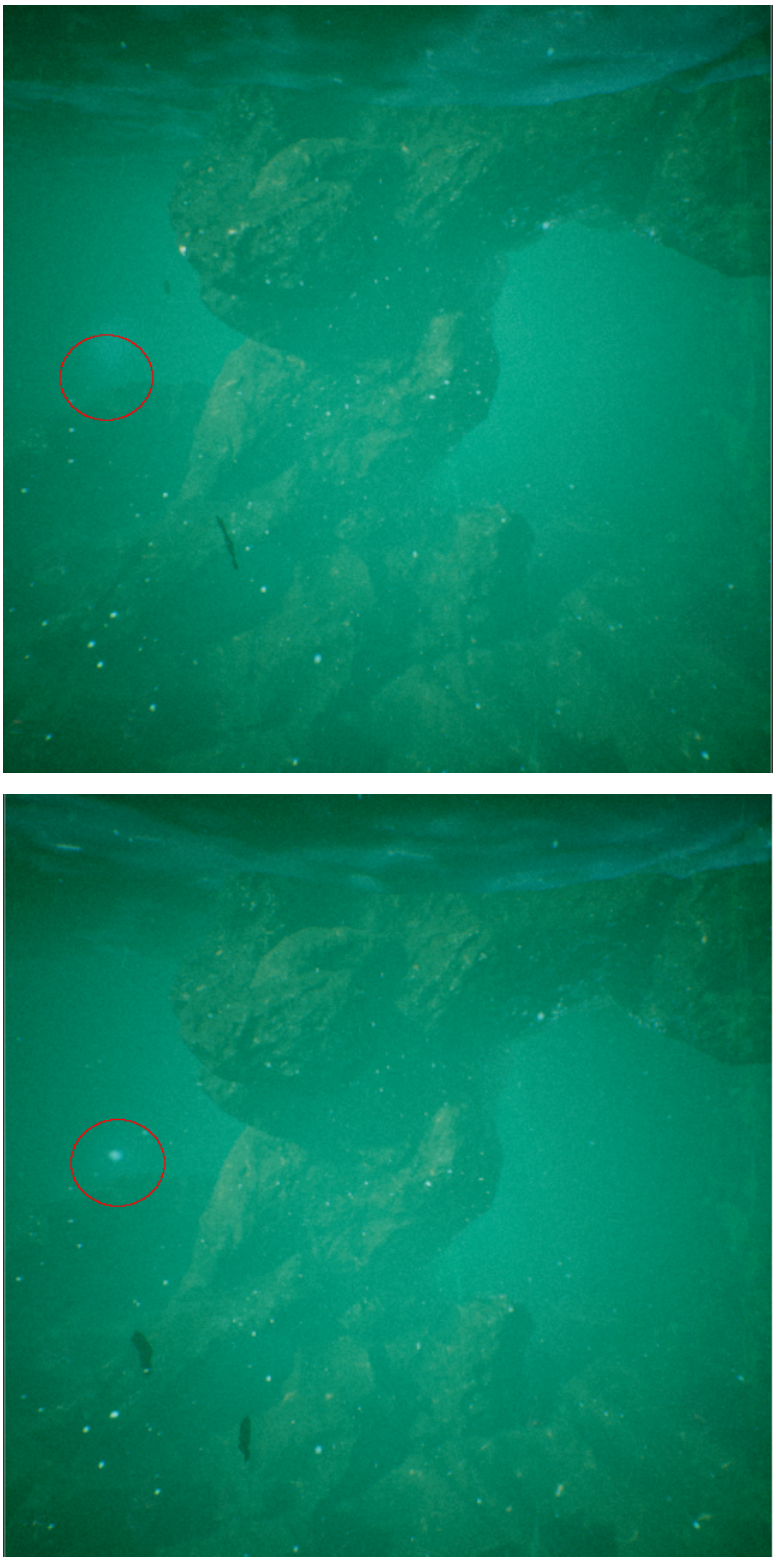


Figure 6.10: Using a plane in front of the camera. In the first image, we didn't use a plane in front of the camera. We can see that the particles with nothing behind them (circled in red) were completely diffused whereas in the second image after adding the plane the particles are visible as we would expect from a real scene.

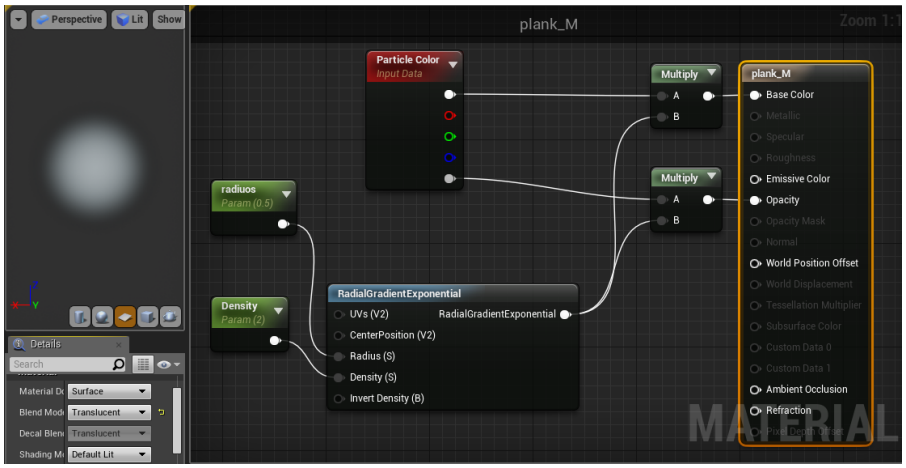


Figure 6.11: A blueprint of the zooplankton material in the Unreal Engine's material editor. The left pane shows the final result and on the right the generator blueprint of the plankton material. The RadialGradient-Exponential component is multiplied by the particle colour (input to the blueprint) and fed as a basic colour an opacity

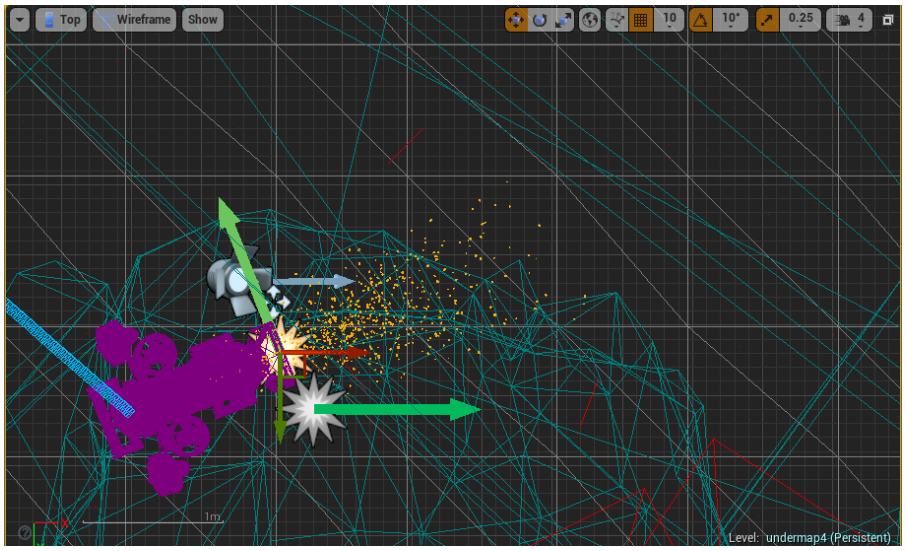


Figure 6.12: Dynamic Emitter Setup. This wireframe image shows an emitter (the particles in Yellow) attached to a ROV (in Magenta) to simulate thruster currents.

The shape and the texture of the plankton can be very complex depending on the type of the organism but since they are still relatively small we use an ellipsoid approximation of the shape. The ellipsoid shape gives us some asymmetry around the middle axes aligned to the world frame which will create an asymmetric backscatter effect. To generate the ellipsoid shape we generated a new material (Figure 6.11) based on a "RadialGriadientExponential" component which basically means that it exponentially reduces the intensity based on the distance from the center. The result is then multiplied by the particular colour and fed as a base colour and opacity. The partial opacity makes rendering of the particle to be even more realistic by simulating the transparency of the plankton. In the particle engine, we choose to generate the particles with different random scaling around the X axis. The random scaling together with random initial orientation will create randomly oriented floating ellipsoids. Figure 6.7 shows the final zooplankton simulation result. In order for the translucent particle to behave correctly in terms of blending with the background, we had to add a plane at a certain distance in front of the camera. In figure 6.9 we can see the top view of this setup. If we didn't add this plane when the particle was not in front of other object, it appears completely diffused (Figure

6.10). The plane was attached to the our simulated camera coordinates in such way that any movement of the camera will result in a movement of the plane respectively. Moreover, the plane is not visible in the scene since it is positioned at a relatively long distance from the camera and it is masked by the simulated fog presented in the previous section.

Figure 6.13: Motion blur. The upper image is a real image (Pelorus Sound New Zealand) taken when we lowered the camera into the water. We can see the short lines blurring patterns of the Zooplankton due to that movement. The lower image is a simulated image. This image was created by rotating the simulated camera and setting the motion blur effect in the unreal engine.

As discussed previously, we can add more than one emitter to have finer control over the simulated scene. We can even add dynamic emitters that move in the scene. An example of such a setup can be seen in Figure 6.12. In that example, we added to an underwater ROV scene, a dynamic emitter with origin behind the camera directed towards the camera by creating directional initial velocities. This kind of emitter simulates the initial movement of a ROV where the thrusters push water together with organic matter towards or away (depending on the direction of the thrusters) from the camera.

We can further improve this model by dynamically changing the velocity of the emitted particles as shown in Figure 6.14. This kind of approach lets us simulate the different movement of the particles depending on the ROV behaviour.

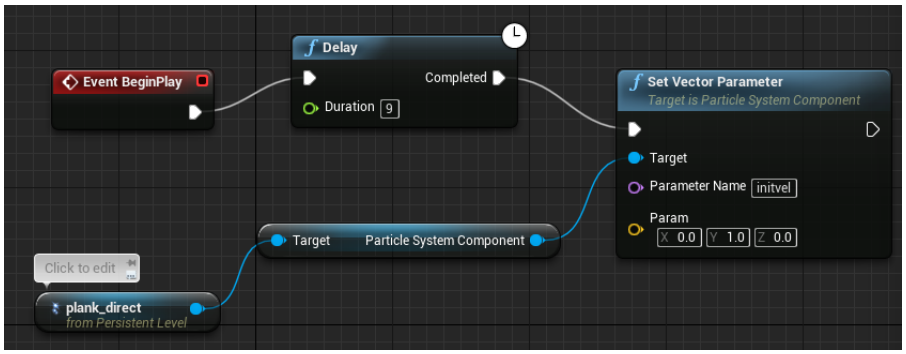


Figure 6.14: Controlling the emitter parameter. This blue print shows an example of controlling the emitter parameter. After 9 seconds from the starting of the simulation, we change the velocity vector of the emitted particles. The parameter "initval" is set inside the particle engine's editor as the initial velocity of the particles.

Finally, to simulate motion blur in computer graphics, usually, multiple images or pixels (or subframes) are taken and averaged [112]. The blurring can affect the whole image in case of camera movement or just moving objects inside the field of view if the camera is stable. In the unreal engine, we can control the intensity of the blurring by adjusting the motion blur amount parameter. In figure 6.13, We can see the upper real image that we took underwater as an example of motion blur of plankton and the lower image is the simulated one done by rotating the simulated camera around the Z axis.

6.5 Conclusions

In this chapter, we discussed the essential concepts and implementation aspects related to the generation of a visually convincing underwater plankton simulation. Different types of plankton and their optical and behaviour characteristics were described in detail. We presented a new approach to the underwater simulation by using a particle engine for the zooplankton simulation and Exponential Height Fog for the phytoplankton simulation. The realism of the images was demonstrated by visual comparison of the simulated images generated using Unreal Engine 4 with real images. We also presented real underwater images to demonstrate the challenges computer vision algorithms or human operators might have when facing underwater imagery. Most of the modelling and design aspects used in our work were based on information gathered by watching and experimenting with real underwater systems. This research which was focused on creating and controlling underwater marine environment can serve as a starting point towards a more in-depth computer vision simulation research. For example, future research can deal with comparing tracking feature experiments with data gathered from an underwater camera and the simulation. In our underwater ROV research, we are aiming towards running localization and navigation tasks which are widely used on a ground and above ground environments. This research is a part of recreating a highly controlled underwater simulated environment and will help us creating high fidelity localization and navigation experiments.

Chapter 7

A Generalized Simulation Framework for Tethered Remotely Operated Vehicles in Realistic Underwater Environments

7.1 Introduction

7.1.1 Background

Remotely Operated Vehicles are being increasingly used in aerial [113], land and underwater applications such as inspection, photography, surveillance and recovery. In most cases such as drones and land vehicles, cameras and other vision sensors like depth sensors are attached to the body to provide real time information, either to a remote operator or to a an autonomous system. The need for visually realistic simulation become clear under those circumstances to provide footage as close as possible to a real environment. Luckily, the advancement in the game engine industry provides the tools for that unique combination. One good example is Microsoft AirSim which is a platform for testing and developing drones with computer vision capabilities [7]. Another example is the "NVIDIA DRIVE CONSTELLATION" a simulated platform with a huge amount of training scenarios for autonomous driving [15]. Our aim is to develop a comprehensive simulation framework with the above capabilities in a highly complex and challenging underwater ocean environment. We will cover all the necessary tools and models we developed to address computer vision problems in an underwater simulated environment.

7.1.2 Importance Of Underwater Simulation

The importance of simulations based on aquatic environments is especially evident under the autonomous systems domain. For air and land vehicles, we can assume some kind of stable wideband communication, such as cellular or satellite communication. That kind of communication gives the remote operators much-needed information during experiments or for monitoring the behaviour of the vehicle in the form of telemetry data and video. Due to the radio frequency propagation characteristics of the air medium, communication and location are usually available. On the other hand in underwater ocean environments, radio frequency is rapidly attenuated which make it impractical for both geolocating and wideband communication. The underwater environment may also not be accessible in case of a malfunction and subsequent reclaiming of a ROV, such that investigating a malfunction may not even be

possible. For those reasons, underwater ROVs are usually connected (tethered) with a cable to a controlling platform above the water.

With increasing applications in underwater exploration, ROVs attached with RGB cameras or even stereo cameras have become more prevalent. The design of ROVs with attached computer vision systems calls for the development of visually realistic simulation models where vision, dynamics and control aspects of the systems can be tested.

A full-scale system simulation that can perform underwater experiments in a fully controlled environment with access to the sensory data and ground truth helps to bridge that gap. Realistic visual effects can bridge this gap even more by providing realistic video for the purpose of autonomous computer vision capabilities. Moreover, visually realistic simulation provides the effects important to accurately recreate effects important in computer vision such as lights shadows refractions visibility and more.

7.1.3 Related Work

Several underwater simulations exist. UWSIM [114]) is an open source project implemented using OSG (Open Scene Graph) as the graphics engine. Another simulation environment used gazebo (A robot Simulation Environment) for the underwater simulation [115]. There are also highly realistic commercial ones like Multi-ROV training simulator [116] which are focused on training personnel for underwater ROV missions. The open source simulations, although they are quite flexible, tend to have less realistic features compared to game engine based frameworks where the commercial ones are driven by market demands which are mainly mission training oriented. State of the art game engines today uses physically based rendering (PBR) which gives us a more accurate representation of materials, camera models and the light interactions compared to the traditional methods, in addition to wide range of tools to build a rich underwater simulation similar to what was achieved by Microsoft AirSim [7].

To validate underwater vision and navigation concepts, researchers usually carry out the experiments in highly controlled environments such as swimming pools [117]. Those environments usually lack the essential effects such as plankton and bubbles [118] that affects the visibility and the overall performance of the vehicle under real conditions.

7.1.4 Our Contribution

This chapter aims to provide all the necessary details including code examples for building a full scale visually realistic underwater simulation. We cover the theory and practical aspects of the mechanical simulation as well as the visual simulation and the different components of the ROV and the simulated environment. We would like to point out our simulation model uses some of the fundamental assumptions such as a regular cuboid with negligible products of inertia and does not include models of robotic arms. These additional characterisitcs can be easily incorporated into the model if required. We also provide the reader a practical test case for building multi-simulation of the popular OpenROV (a real underwater ROV) which incorporates components from different domains. Real computer vision algorithms were tested and compared under our simulation and using a real ROV to validate the proposed approach. Our framework for generating visually realistic underwater simulation is fully open sourced, based on a game engine and includes additional simulation components for the final purpose of computer vision research.

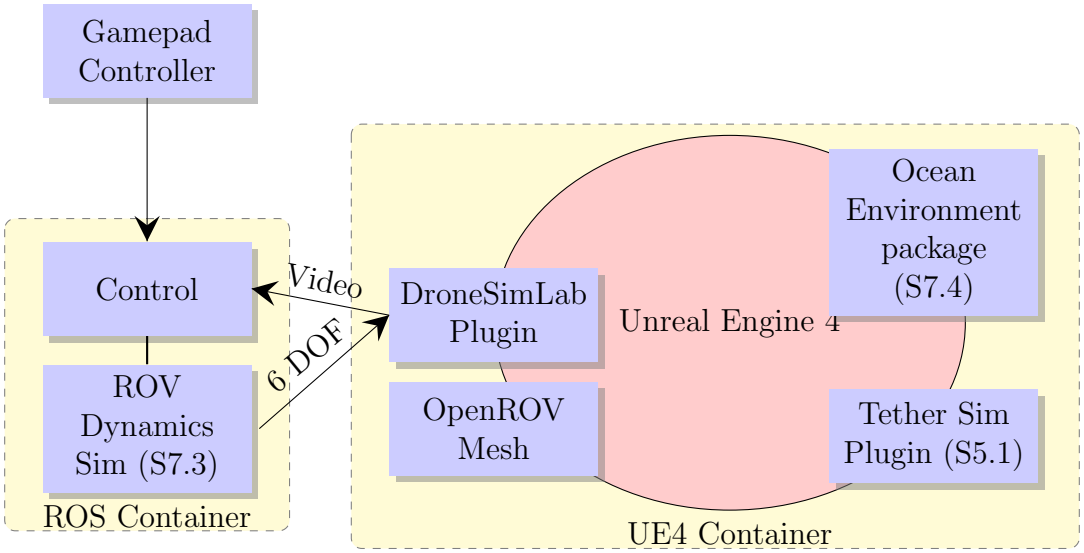


Figure 7.1: Simulation Diagram. This diagrams shows the components of the simulation and the main communication channels. S5.1,S7.4 and S7.3 refers to the relevant sections/chapters in this work.

7.2 Simulation Framework Overview

The multi-simulation framework comprises of several sub simulation components and are implemented using different methods and tools as shown in Figure 7.1. The components are all linked together using the DroneSimLab framework [36] and this simulation is currently a part of the published demos and can be downloaded as part of the framework. Figure 7.1 shows the main components of underwater simulation. A gamepad controller is used to control the ROV. The input from the gamepad is then mixed in the control module to produce the relevant thrust from the thruster. For example, if the user is pushing forward then the controller is increasing evenly the power to the back thrusters. In contrast, if the user is pushing to the right only the power of the left thruster increased. The output of the control which is the thrust power of each thruster is then fed into the ROV dynamics module which in turn is responsible for providing 6 DOF (degrees of freedom) of the position and orientation of the ROV. The 6 DOF data is then fed into to the Unreal Engine environment through the DroneSimLab plugin which updates the position and the orientation of the OpenROV Mesh.

7.3 Simulating ROV Dynamics

To simulate the ROV dynamics two approaches may be taken. The first is to use the internal game engine physics engine. The second is to use external engine independent of the game engine environment. The second approach decouples the dynamic engine from the game engine environment which allows freedom in choosing tools for the dynamic simulation as well as a different game engine. The second approach although presenting significant robustness can only work in case of limited interaction between the ROV and the environment or mostly unconstrained motion. This means that it might be suitable for drones and underwater ROVs but not for land vehicles who have constant interaction with the simulated game engine environment. Land Vehicle motion is constrained by the road and needs constant input regarding the normal's surfaces that come in contact with the vehicle. For our purposes, we used the second approach since most of the ROV interaction is with the surrounding waters and that is mostly independent of the ROVs position. If we increase the ROV interaction with the environment for example by adding a robot arm, we will need to reconsider that approach.

In order to simplify our problem we made some assumption which will affect the ROV

behaviour and need to be consider depending on the goal and the required accuracy from the simulation. Since our main focus of the simulation was the visual aspect, some simplifications were made in the mechanical domain, which will be covered in the subsequent subsections.

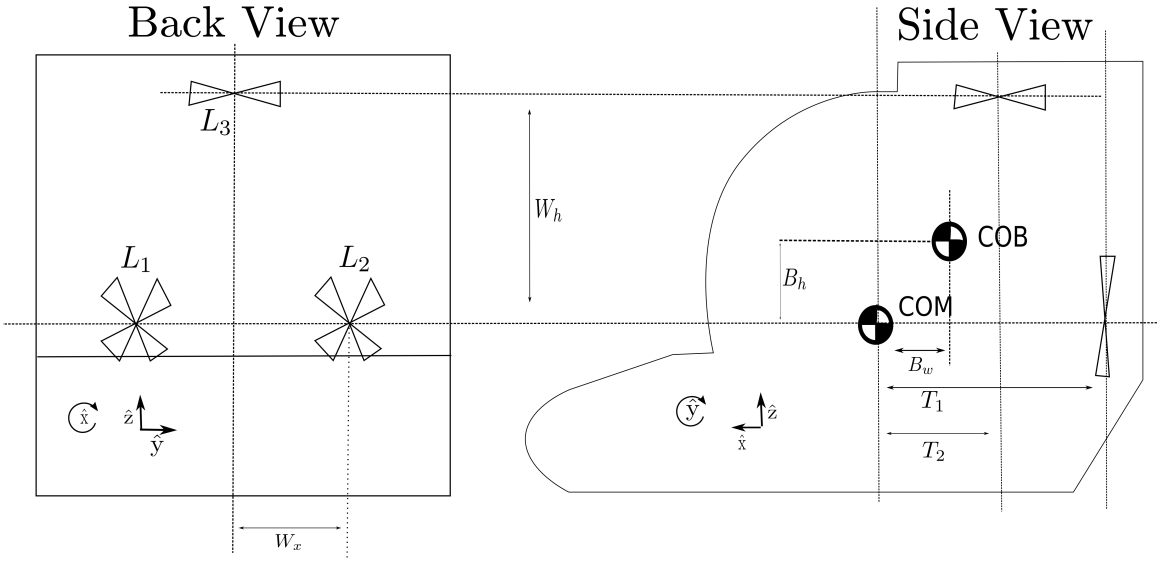


Figure 7.2: Schematic drawing of the OpenROV where:
m_A: Mass of the ROV.
T_1: Distance from COM (Center Of Mass) to back thruster axis.
T_2: Distance from COM to upper thruster axis.
W_x: distance from center axis to back thruster axis.
W_h: distance from center axis to upper thruster.
B_h: the z measure of COB (center of buoyancy) distance from COM
B_w: the x measure of COB distance from COM.
$L_{1..3}$: Position points of the thrusters

7.3.1 Method Overview

The main tool we used to implement the ROV dynamics was SymPy Mechanics package and more specifically the Kane's method implemented in this package [119]. SymPy is a Python library for symbolic mathematics. It is a full-featured computer algebra system [120]. We use the symbolic tools to define the necessary inputs for this method. A gives the necessary implementation details, the code and the numerical values we used for the following method.

Kinematics

For inertial reference frame N and a rigid body B of the ROV, we define the generalized coordinates $q_{0..2}$ and $q_{3..5}$ as the ROV position and orientation respectively and the subsequent generalized speeds which are the linear and angular velocities as $u_{0..2}$ and $u_{3..5}$ respectively. To properly define the ROV reference frame R we define the order of rotation to be zyx intrinsic rotation order which sometimes referred as yaw, pitch and roll rotations. The rotation is done in 3 stages and defines 3 more reference frames which will also be referenced later as we define more components of the simulation. First, we define a reference frame Φ which is rotated around the z axis of N by an angle q_5. This is done using the functions "orientnew" and "set_ang_vel" from SymPy mechanics. We repeat this process and define Θ which is rotated with respect to Φ around the y axis of Φ and finally we define Ψ (the final ROV frame) which

is rotated around the x axis of Θ. We can use the power of symbolic to display us the rotation matrix as presented in Eq. 7.1.

$$\begin{bmatrix} \cos(q_4)\cos(q_5) & \sin(q_5)\cos(q_4) & -\sin(q_4) \\ \sin(q_3)\sin(q_4)\cos(q_5) - \sin(q_5)\cos(q_3) & \sin(q_3)\sin(q_4)\sin(q_5) + \cos(q_3)\cos(q_5) & \sin(q_3)\cos(q_4) \\ \sin(q_3)\sin(q_5) + \sin(q_4)\cos(q_3)\cos(q_5) & -\sin(q_3)\cos(q_5) + \sin(q_4)\sin(q_5)\cos(q_3) & \cos(q_3)\cos(q_4) \end{bmatrix} \quad (7.1)$$

In order to apply gravity and buoyancy forces we need to define the COM (Center Of Mass) and COB (Center Of Buoyancy) in the reference frame. The COM point is defined with respect to the world coordinate origin and are simply $q_{0..2}$ and the velocity is $u_{0..2}$ in N. The other points on our rigid body are defined with respect to that point. the position of COB is defined as:

$$P_{cob} = B_w \hat{\mathbf{\Psi}}_{\mathbf{x}} + B_h \hat{\mathbf{\Psi}}_{\mathbf{z}} \quad (7.2)$$

Where B_w and B_h are the dimentions presented in Figure 7.2 and $\hat{\mathbf{\Psi}}_{\mathbf{x}}$ and $\hat{\mathbf{\Psi}}_{\mathbf{z}}$ are the unit vectors of our ROV frame.

To calculate the velocity of COB in N we use the folowing eq. [121]

$${}^N\mathbf{v}_{cob} = {}^N\mathbf{v}_{com} + {}^N\omega^{\Psi} \times \mathbf{r} \quad (7.3)$$

Where ${}^NV_{cob}$ and ${}^NV_{com}$ are velocities in the inertial frame N and ${}^N\omega^{\Psi}$ is the angluar velocity of Ψ with respect to N and $\mathbf{r}$ is the position vector from COM to COB. In a similar manner we define velocities of the rest of the fixed points in the ROV frame which represent the positions of the thrusters eg L_1,L_2 and L_3 as can be seen in figure 7.2.

Eq. 7.4 is enforcing the relationships between the generalized coordinates and the generalized speeds and referred as the kinematic differential equations which are input to the Kane's method implementation in SymPy mechanics package we are using [119] [122].

$$u_r = \dot{q}_r \quad , r = 0..6 \quad (7.4)$$

Inertia

As an input to the Kane's method [119] we need to calculate the inertia dyadic or matrix. In our simulation we simplify the problem to the calculation of a rectangular box which in turn can be calculated by the following equations:

$$I_x = \frac{1}{12} m(y^2 + z^2) \quad (7.5)$$

$$I_y = \frac{1}{12} m(x^2 + z^2) \quad (7.6)$$

$$I_z = \frac{1}{12} m(x^2 + y^2) \quad (7.7)$$

where x, y and z are the dimensions of the box along the corresponding axis, and m is the mass of the rigid body. I_x , I_y and I_z are the moments of inertia along the corresponding rotating axis. The products of inertia in that case are all zero due to the symmetry of the box e.g:

$$I_{xy} = I_{xz} = I_{yz} = 0 \quad (7.8)$$

Dynamics

The next step after kinematics is the dynamics which deals with the forces applied in our system. Under the simplified assumption of uncompressible fluids and non turbulent flow and e.g. small Reynolds number the drag force is proportional to the velocity and can be written as:

$$\mathbf{F_D} = -\mu\mathbf{v} \tag{7.9}$$

where the F_D is the drag force, μ is a term representing the characteristics of the fluid and the shape of the object [123] and $\mathbf{v}$ is the linear velocity of the body. Under the same assumptions and in a very similar manner we define the rotational hydrodynamic drag torque as follows:

$$\mathbf{T_D} = -\mu_r\boldsymbol{\omega} \tag{7.10}$$

where μ_r is the rotational drag term representing the shape of the object and the surrounding fluid and ω is the angular velocity for a given axis. μ and μ_r can be estimated either analytically by looking at known solutions of simple objects like spheres and cylinders, dedicated software simulation tools or empirically depending on the goal of the simulation. From our perspective, although the final goal of the simulation is to generate visually realistic scenarios, adding some sort of drag is necessary and cannot be neglected in order to generate realistic maneuvers of our simulated ROV.

In addition to the hydrodynamic drag we need to add the constant forces, the gravity and the bouncy forces f_b. The gravity acts at the COM point and the buoyancy at the COB point. The buoyancy is calculated from the displacement volume. For controlling the ROV we use 3 thrusts which generates 3 externally controlled forces at points L_1,L_2 and L_3 (Figure 7.2). Though it is not negligible in small ROV, the thrust can also generate a torque and for simplicity it was not added to the simulation. The next equations 7.11,7.12 and 7.13 summarize the pairs of points and the forces applied to theses points.

$$ControlForces = \{L_1, f_1\hat{\mathbf{\Psi}}_\mathbf{x}\}, \{L_2, f_2\hat{\mathbf{\Psi}}_\mathbf{x}\}, \{L_3, f_3\hat{\mathbf{\Psi}}_\mathbf{z}\} \tag{7.11}$$

$$ConstantForces = \{P_{com}, -gm_b\hat{\mathbf{N}}_\mathbf{z}\}, \{P_{cob}, f_b\hat{\mathbf{N}}_\mathbf{z}\} \tag{7.12}$$

$$LinearDamping = \{P_{com}, -\mathbf{v}\mu\} \tag{7.13}$$

Equation 7.14 summarizes the torque and the reference frame applied on the ROV

$$DampingTorque = \{\Phi, -\mu_r(u_5\hat{\mathbf{N}}_\mathbf{z} + u_4\hat{\mathbf{\Phi}}_\mathbf{y} + u_3\hat{\mathbf{\Theta}}_\mathbf{x})\} \tag{7.14}$$

Kane's Method

Kane's method forms the following expression for a given reference frame N and a system S:

$$\tilde{F_r} + \tilde{F_r}* = 0, r = 1, ..., p. \tag{7.15}$$

where p is the number of degrees of freedom in N frame and $\tilde{F_r}$ and $\tilde{F_r}*$ are respectively the nonholonomic generalized active forces and the nonholonomic generalized inertia forces for S in N [124]. Eq. 7.15 also known as Kane's dynamic equations and can be rearranged to the following form which is also implemented in SymPy mechanics package [125] [126]. We refer to the Kane's implementation as a "black box" which the implementation details are out of the scope of this thesis. The previous loads e.g. the forces and torques , the generalized coordinates and speeds, The Kane's differential equations are fed into this "black box" which according to Kane's method forms the following differential equation.

$$M(q,t)\dot{u} = F(q,u,f,t) \tag{7.16}$$

where $\dot{u}$ is a $p \times 1$ matrix having the time-derivative $\dot{u_r}$ of the generalized speed $u_r(r = 1, ..., p)$, M is a $p \times p$ matrix who's elements are the generalized coordinates q and the time t and F is a $p \times 1$ matrix who's elements are the generalized coordinates q, the generalized speeds u, f is the input external forces and the time t. This form enables us iterative integration to calculate the generalized speeds and the generalized coordinates for each step in the simulation. In the SymPy package, the M matrix is called "mass matrix" and the F vector is called the forcing vector. This form of differential equations allows us to iteratively integrate the position and orientation of the ROV with respect to time.

Dynamic Simulation Results

To test the simulation we generate a demo scenario. We put the ROV underwater without applying any external thruster force for 20 sec. In terms of the simulation this meas that the initial integration conditions of the generalized speeds and coordinates are set to zero. The next 20 seconds hence the second stage. We apply uneven thrust force for thrusters 1 and 2 (described in Figure 7.2).

In Figure 7.3 , we can see that in the first stage since the ROV is positioned with 0 pitch angle and since the COB point is not directly above the COM point, the ROV is oscillating around the y axis. The oscillation are dampened due to the dampening drag forces described in the previous subsections. In the next stage we can see that the ROV is turning due to the uneven thrust forces applied. The ROV is also in a dive since it was stabilized in a positive pitch as can be seen in sub figure b.

7.3.2 Conclusions

We have presented a step by step procedure for creating dynamic rigid body underwater ROV simulation. The output of this process is formulated in eq 7.16. This form of representation decouples the model developed from the actual simulation. This means that we can use the output ("Mass matrix" and "Forcing vector") and integrate it into any simulation framework independent of programing language or operating system. In addition, this method is highly configurable. For example, We can easily add an additional thruster and position it in the ROV frame. We can test the new configuration in different scenarios as demonstrated in the previous results. We presented the process of integrating the simulation products in the final full-scale 3D simulation. Some assumptions we made in order to simplify the simulation. For example, we ignore Coriolis forces due to the low relative speeds in which underwater ROVs are moving, having said that modifying the simulation in that vector is achievable.

7.4 The Underwater Ocean Environment

Simulating visually realistic marine environment can be challenging due to the presence of several factors affecting the illumination and motion of objects within a region. In contrast to the air medium, the underwater marine environment appears significantly more turbid and dynamic, being rich with organisms. Each organism has its own special visual characteristics and dynamics. They can be for example, semi-transparent like jellyfish or completely opaque. The scene can be highly dynamic where schools of fish pass very close to the camera. Such complex and dynamic environments need to be modelled for simulating the motion of an underwater remotely operated vehicle (ROV). This section focuses on generating the underwater

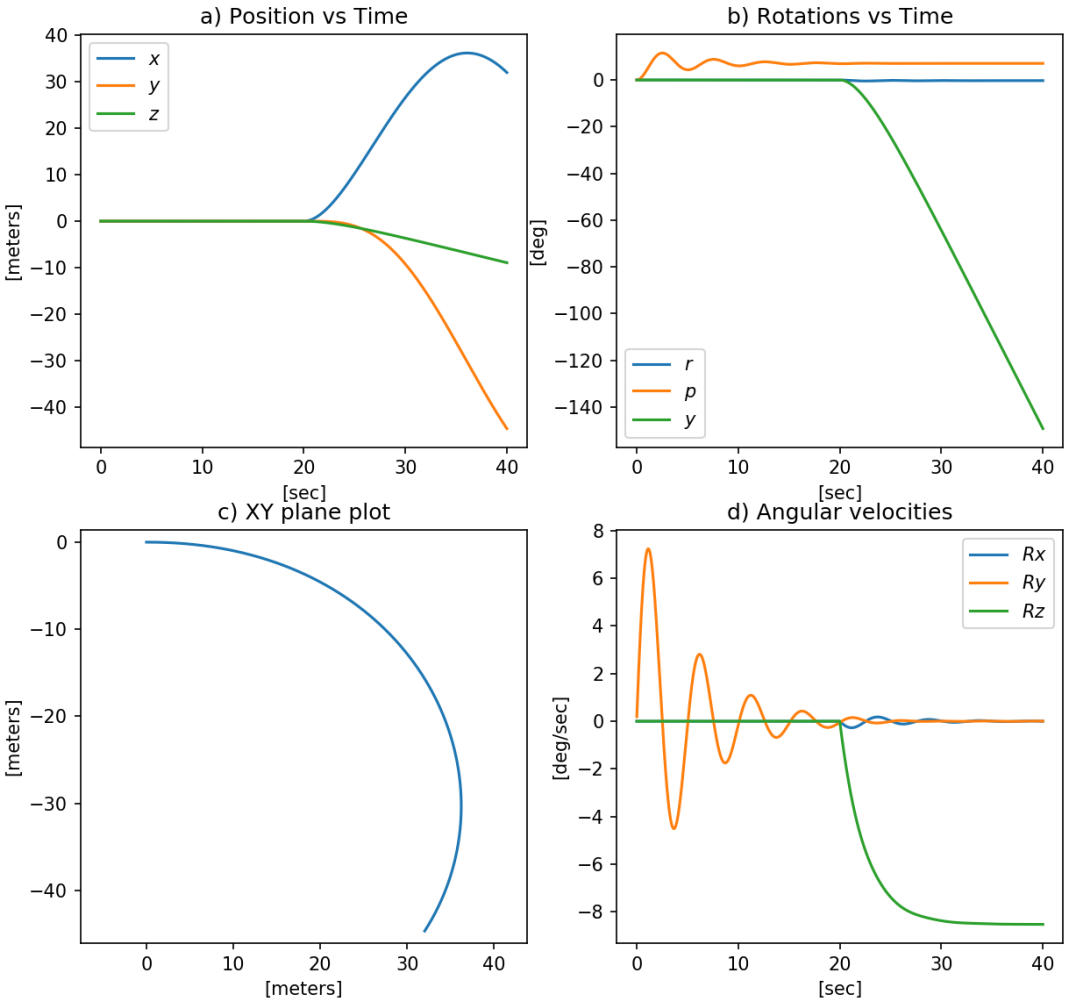


Figure 7.3: Simulation result summery. a) shows position over time. b) shows rotations over time using Euler angels Yaw,Pitch and Roll. c) a projected course of the ROV in the X-Y plane. d) shows the angular velocities around the X,Y and Z axes.

ocean environment and in the marine environment the associated visual effects. The generation of bioluminescent plankton which is a significant part of the environment was covered in chapter 6. Today the visual capabilities of game engines are exceptional. Those capabilities are motivated by a multi-billion industry and the rapid advancements in the graphics hardware domain. In our simulation which focuses on the realism of the environment, using game engine is a natural choice.

7.4.1 Generating Marine Environment in a Game Engine

From a research perspective, working with a game engine as the provider of the visual world is similar to finding or creating a real-world scene for the purpose of doing an experiment. For example, going outside and find a suitable place for an outdoor experiment is parallel to the process of searching an existing environment in a game engine marketplace. In a Game Engine marketplace, we can find an existing environment that we can manipulate for our purposes. This process needs to be done carefully since environments taken from a marketplace engine might have some unreal effects for gaming and entertainment purposes. In our case, we took an existing underwater marine environment with existing rocks, fish, relics etc [127]. Figure 5.7 shows such a rich environment. We can see in this image some rocky structure like an underwater rock gate, some fish and also our simulated robot maneuvering. For a more complete visual simulation we added to it some other needed components such as the ocean plankton which we will discuss thoroughly in chapter 6. The end result (as can be seen in the attached videos) that we got was a convincingly realistic underwater environment which we could control and manipulate as needed. We can control and move any object in the scene and for some objects we can change some key characteristics to simulate different conditions. Figure 7.4 shows a seaweed simulation taken from the package [127]. The seaweed which has a root planted on the sea floor is free to move according to the ocean currents. The seaweed component simulation is based on the unreal engine SimpleGrassWind which when applied underwater gives similar effect [128]. We can control the amount of wind and type by changing the relevant blueprint.

Figure 7.4: A sea weed dynamic simulation based on the “grass wind” asset from UE4

7.4.2 Simulating The Camera

The intrinsic params of the camera have a large impact on the behaviour and performance of a computer vision system. For the purpose of simulating the camera, we are using the SceneCapture2D component which is used for the purpose of rendering the scene into textures. Game engines use this component to simulate scenes with a screen like objects like security cameras and monitors for example. In this subsection, we are covering the main adjustments we made to the physically based camera model of the Unreal Engine SceneCapture2D component to the underwater environment.

The first adjustment was the FOV. Since we are modelling underwater vision conditions we need to update the camera model underwater. For simplicity, we assume a perfect pinhole camera model without distortion. As an input to the game engine, we need to choose the horizontal FOV (field of view). from Snell's law we know that:

$$\frac{\sin\theta_2}{\sin\theta_1} = \frac{n_1}{n_2} \tag{7.17}$$

where each θ is the angle measured from the normal perpendicular to the boundary and n is the refractive index of the respective medium. In our case our ray of light is moving through water and then to air (we ignore the effects of the port) and we get magnification of 1.333. From Eq. 7.17 we can calculate the effective FOV underwater as described in Figure 7.5 and summarized in Eq. 7.18.

$$FOV_{water} = 2\arcsin\left(\frac{n_1}{n_2}\sin\left(FOV_{air}/2\right)\right) \tag{7.18}$$

The Unreal Engine uses physically based post process effects which we can use to better represent the under water conditions. One aspect was the adding of a dirt mask on top of the camera lens. It is quite common in an underwater environment to have dirt accumulated on the port edge facing the water. This effect depending on the magnitude of the accumulation can be significant and may affect computer vision algorithms such as tracking. Figure 7.6 shows a dirt mask pattern (From [127]) we applied in Unreal Engine on our camera component. The dirt mask is blended with parameterized intensity to the camera image.

For additional image noise we added grain noise filter which is also common under low light conditions. Adjusting the grain filter can be done directly in the game engine and we can control the jittering and the intensity.

Another post process effect is the chromatic aberration that simulates the color shifts in real-world camera lenses. The effect is most noticeable near the edges of the image and in relative large FOV angles lenses. The magnitude of the effect is parametrized and experiment can be made to explore the different effects of different values on computer vision algorithms.

We can also simulate the depth of field. Unreal Engine supports cinematic methods and it is aligned with common camera options. In our underwater simulation we didn't focus on that effect (in terms of testing) since the outcome of this effect has some overlap with the fog effect which was more significant in our simulation.

7.4.3 Test Case Results

For the purpose of analyzing the contribution of the simulated environment, we created a test case algorithm based on the ORB future detection algorithm [129] and optical flow tracking based on [59] implemented in OpenCV. In this experiment, we show that the futures extracted from the scene are not the usually the intended futures we would expect and would have an adverse impact on algorithms like SLAM and obstacle detection which are based on tracking features. In Figure 7.7 we can see that tracking algorithm treats the dirt bubble as legitimate

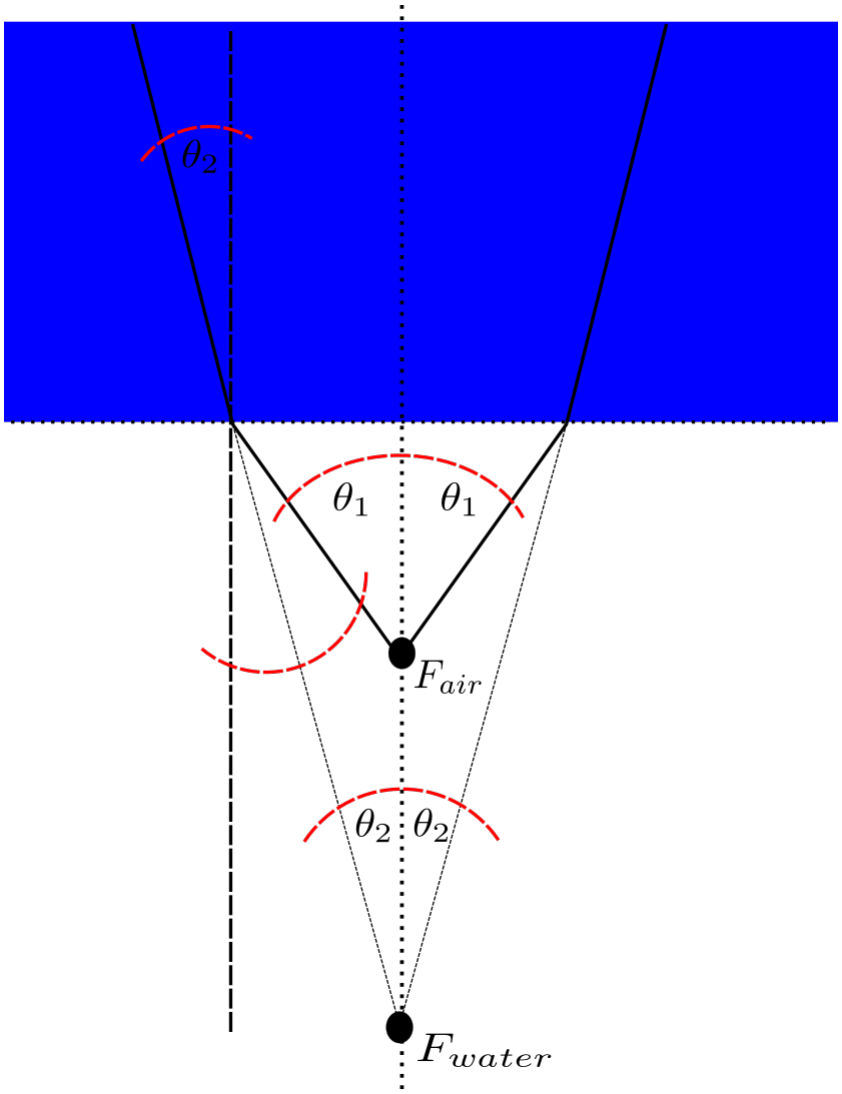


Figure 7.5: Effective FOV underwater. a ray is coming from an angle of θ_2 is refracted with an angle θ_1 which moving the Focal point F_{air} to the effective F_{water}

objects in the scene based on those features there is no camera movement between the frames where in fact we can see (from other features and the objects) that the view is moving upwards. In Figure 7.8 we can see the same effect repeating in the simulation as we intended. In the real image Figure 7.9 which we took underwater, we can see features detected in the view not only on the clear visible objects like the ropes but also on the zooplankton and organic material surrounding those ropes. Similarly in the simulated image in Figure 7.10 we can see features detected on our simulated zooplankton. Since this is a simulated scene we can turn on and off the zooplankton and run the feature detector as presented in Figure 7.11. In This Figure, we can see that features were detected only on the visible object. We can also see that futures were not detected on the distant objects due to simulated fog which reduces the visable gradients.

7.5 Conclusions

In conclusion, this test case experiment sums up the essential concepts and implementation aspects related to the generation of a visually convincing underwater environment simulation. There are many more graphical effects which could not be covered by the scope of the thesis and in our simulation but based on our real underwater experience we tried to cover the main effects influencing our underwater scene. The realism of the images was demonstrated by a

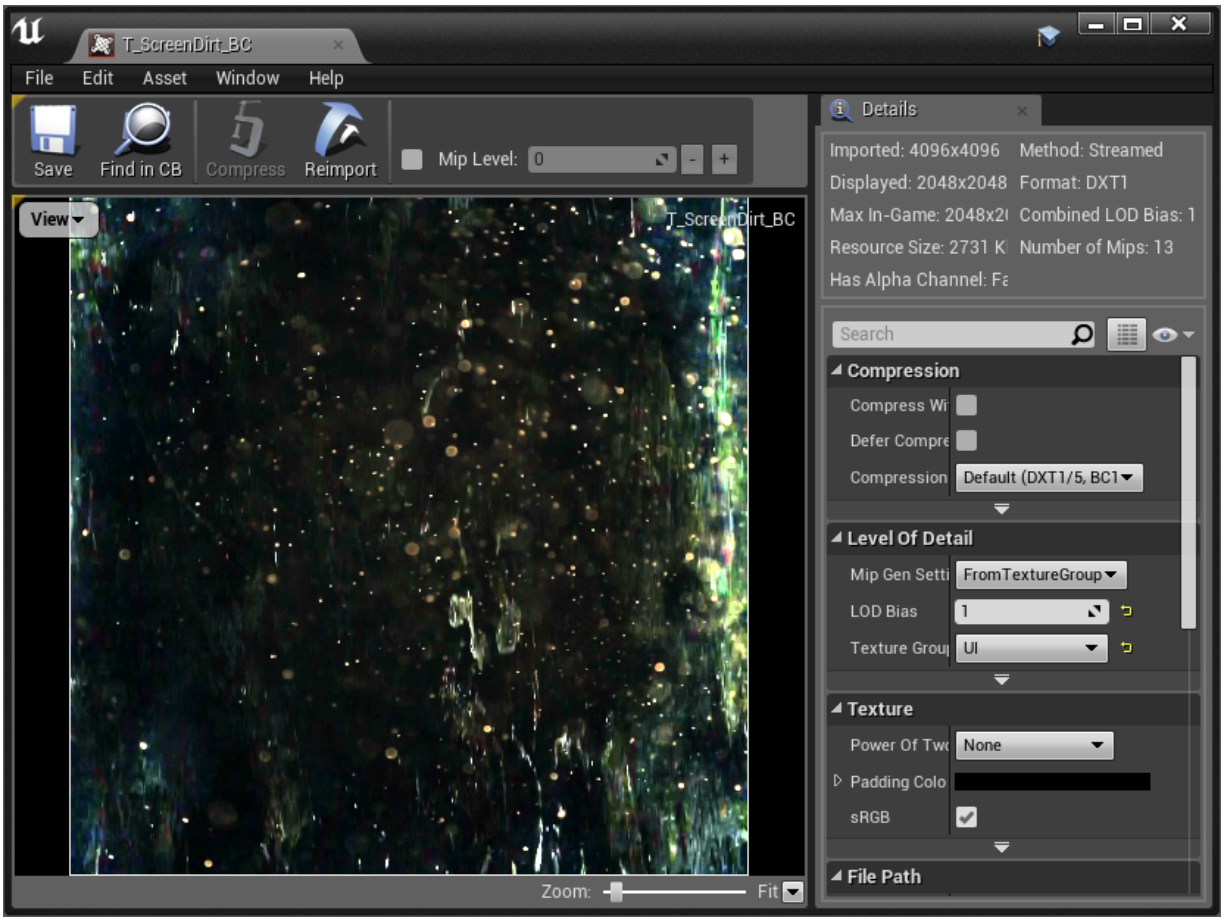


Figure 7.6: Dirt mask added to the simulation. This dirt mask was added to simulate dirt accumulated on the camera port.

visual comparison of the simulated images generated using Unreal Engine with real images. Most of the modelling and design aspects used in our work were based on information gathered by watching and experimenting with real underwater systems. We created a computer vision test case algorithm and compared side by side the performance of the algorithm on real images and videos taken during our real test dives with simulated experiments. The outcome showed a convincing similarity between the simulation and the real environment for feature extraction and tracking experiments.

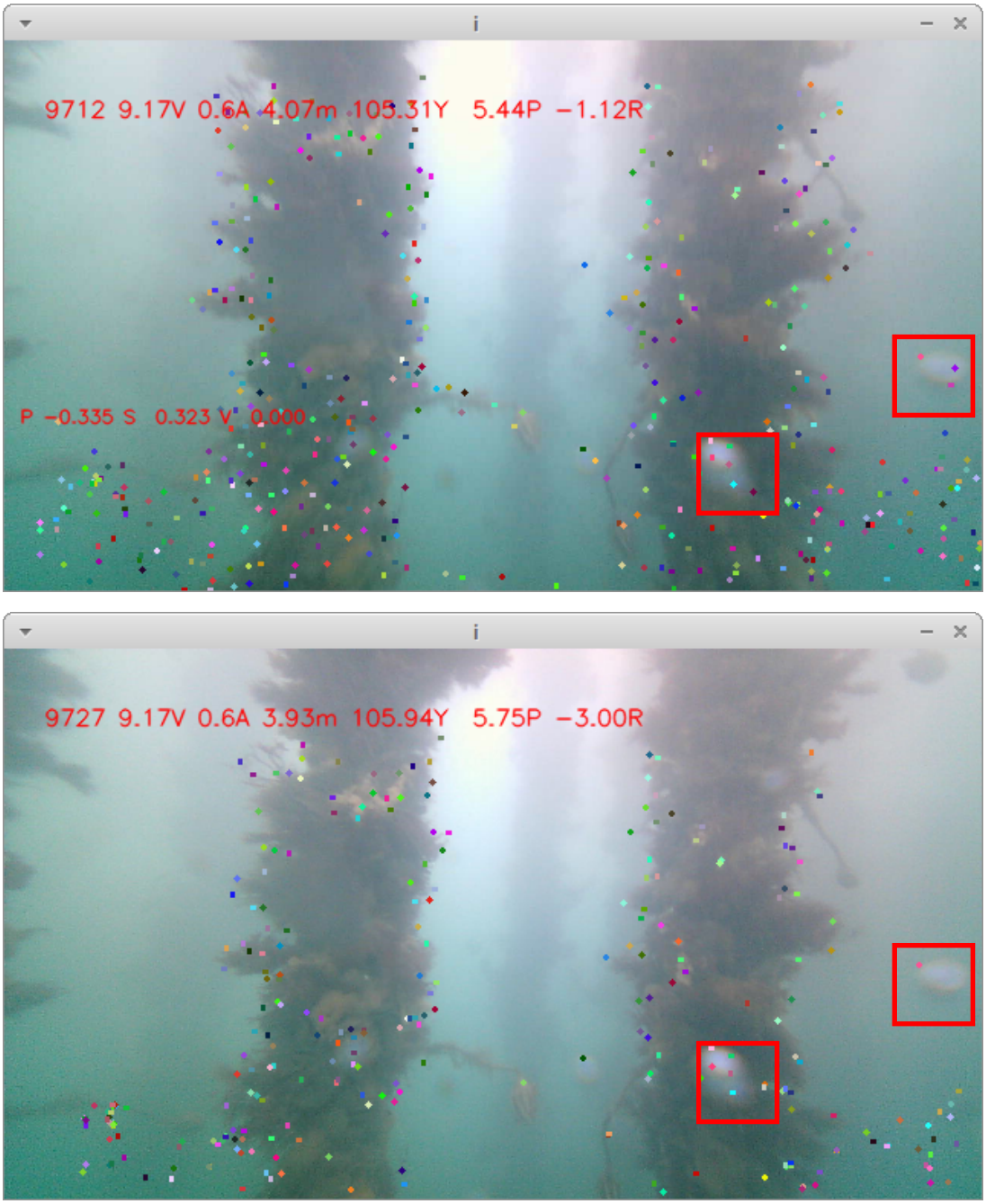


Figure 7.7: Tracking Experiment. This experiment was a real experiment done using the OpenROV platform in a mussel farm near Port Levy New Zealand. Two frames from a tracking experiment, the first number in the telemetry line (upper left corner of each frame) is the frame number. the features in the frames are colored with different colors in order to keep track. Marked in red square are dirt bubbles attached to the camera port.

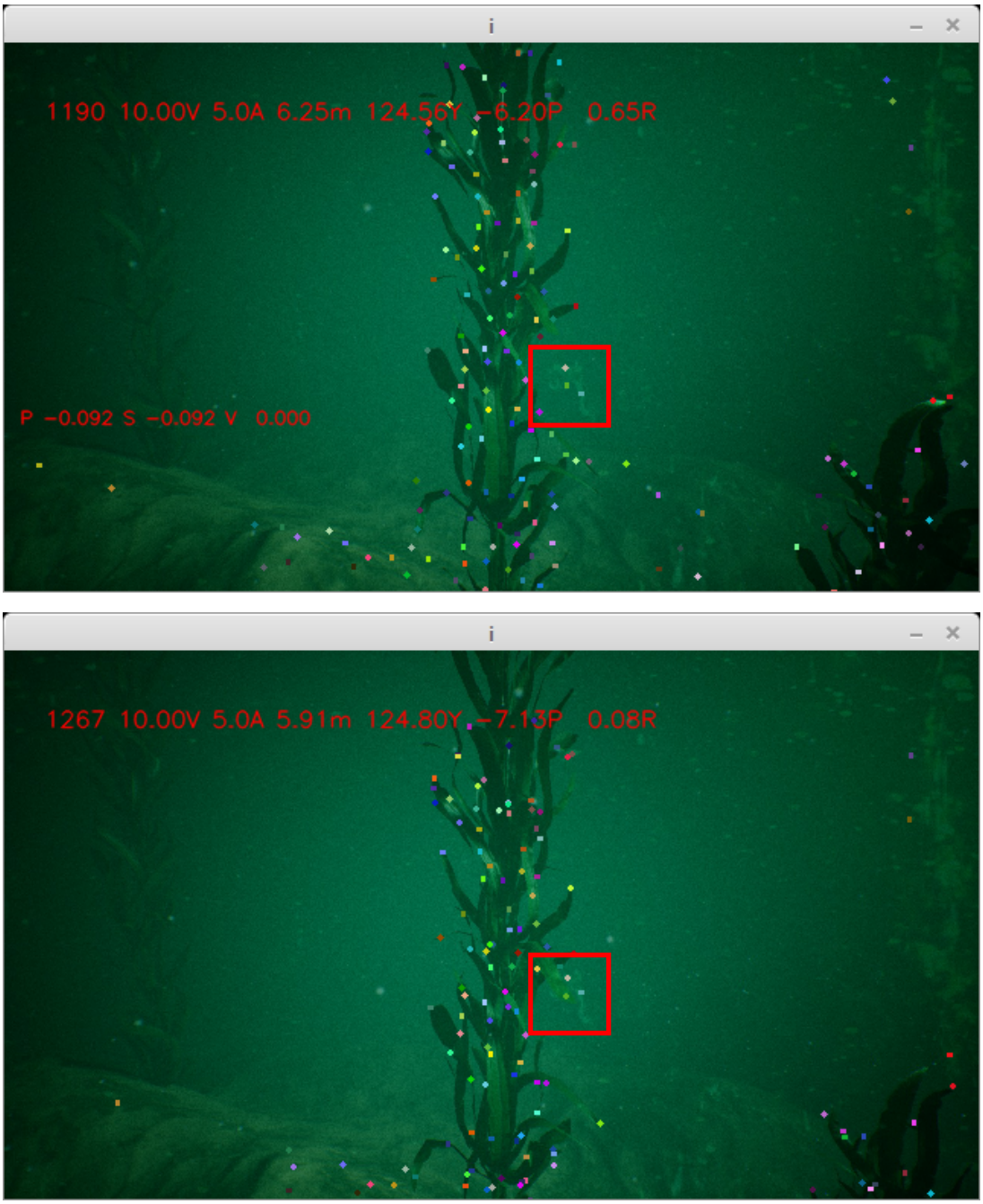


Figure 7.8: Tracking dirt mask. Repeating the experiment done in figure 7.7 in the simulation. Similar to 7.7 we can sea that features are detected and tracked on the dirt mask (mark in red square).

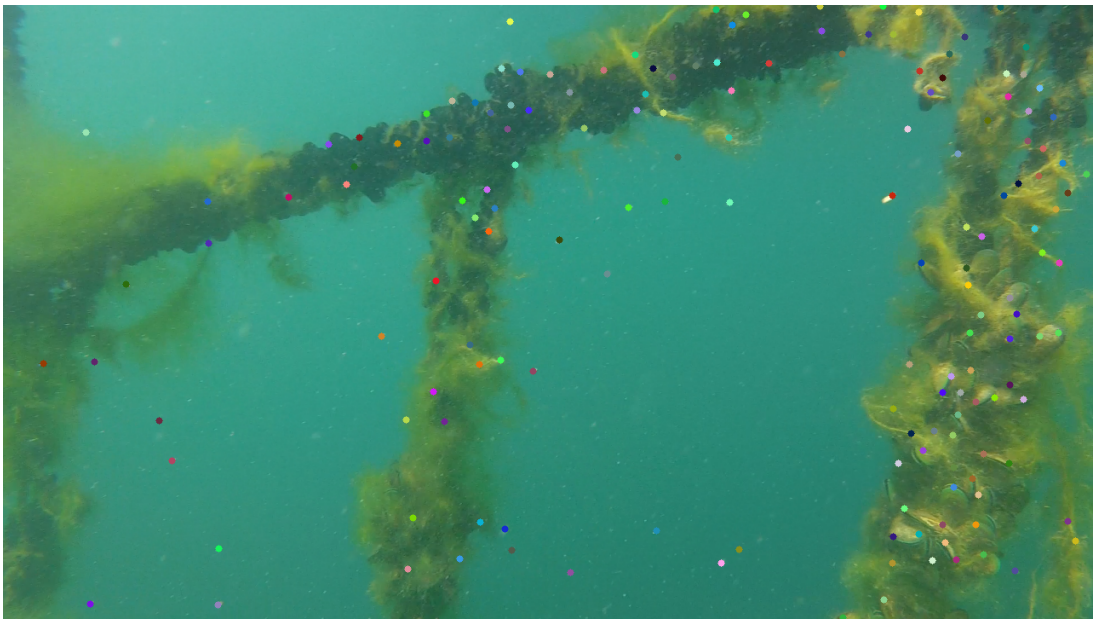


Figure 7.9: Running feature extraction using ORB detector. This real image we took in a mussel farm in marlborough sounds New Zealand. We can see that the features (marked with different colors) were not necessarily detected on the mussel rope but rather on the surrounding organic material.

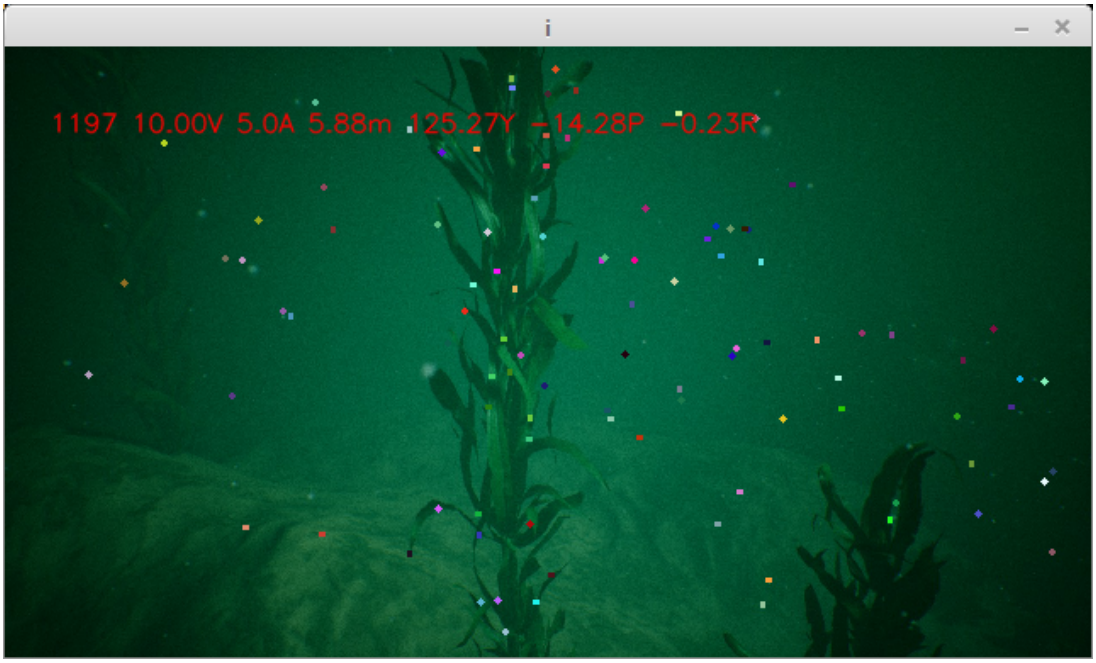


Figure 7.10: Feature extraction in a simulated image with zooplankton but without dirt mask.

Figure 7.11: Feature extraction in a simulated image without zooplankton and without dirt mask. This image shows that as expected no feature where detected on the surrounding medium but only on the front sea weed.

Chapter 8

A Stereo Vision Based Tracking and Localization Technique for Underwater Navigation

8.1 Introduction

Vision-based ROVs are becoming cheaper and more affordable. The prevalence of ROVs can therefore be seen in both recreational and commercial applications. Vision robots can replace human divers. They are more resilient and can dive deeper and for a longer period of time especially in applications focused on vision. Examples of such applications can be inspections of underwater structures such as bridges, dams and ship hulls, all currently performed by human divers. Stereo vision can be used for both localization and reconstruction and a lot of implementations such as various types of SLAM algorithms were developed during the last decade. Those kinds of applications are widely used today as part of autonomous systems like self-driving cars, where some assumptions can be made about the surrounding environment. For example, we assume that most of the environment is static and only the robot is moving. We can also assume that we can see most of the objects in the field of view. That sort of assumptions are not necessarily true underwater and we probably need to apply some other techniques specifically designed for underwater environments, as discussed in later sections. In a lot of cases, the visibility is very limited. On most systems, sensor fusion is necessary and we cannot rely on vision alone due to the reasons mentioned. That is especially true for the underwater world. Complex combinations of sensors and filters can be used to generate a reliable localization.

8.1.1 Research Context and Importance

Controlling an ROV with high quality stereo vision capabilities for the purpose of doing underwater inspections (ship hulls , wharf pylons, bridges etc.) is not an easy task. ROVs, especially smaller ones are susceptible to effects from waves, vortexes and ocean currents. In this chapter we present a reliable and efficient vision algorithm to help a human operator controlling the ROV under those challenging conditions. The most basic maneuver for ROVs is maintaining a 3D position. This is very similar to the basic maneuver of drones maintaining 3D position in air without any control inputs from the pilot. This maneuver reduces the load from the pilot who can for example focus on other tasks such as inspection. We also consider this maneuver a first step towards fully maneuverable auto pilot and maybe in the future a part of a vision based Autonomous Underwater Vehicle (AUV).

In this chapter we will present a method for stabilizing an ROV in front of an underwater

object regardless of drifting currents. This is a very basic and extremely computationally efficient and useful maneuver especially in cases where the ROV does the inspection and we want it to remain stationary relative to a moving point of interest. Surprisingly this task is not easy for a human operator due to currents, buoyancy and other forces we mentioned earlier affecting the ROV. Although, visual odemetry methods were tested for the underwater domain, our focus will also be on the more visually challenging underwater environments in terms of visibility and dynamics. An example of those challenging conditions can be found in figures 8.1 and 8.2. We applied feature extraction algorithms in order to demonstrate the difficulties in running common vision navigation techniques such as SLAM which are based on feature extraction and feature matching. In addition to the fact that our object of interest might be blurred or obscured by organic matter, the landmarks created by the detection of the organic matter are highly dynamic and influenced by the ocean currents or even the currents created by the movement of the thrusters (if an ROV is being used) which makes those features poor landmarks. In addition, range measurement using disparity map results in a highly noisy image with false surfaces due to close and semitransparent particles. Figure 8.3 shows our attempt to apply stereo matching algorithm for the purpose of detecting range [130]. Modifying the parameters of this algorithm did not change the overall outcome.

In this chapter we present results for two types of experiments. The experiments are end to end closed loop experiments which means that we test our method as an integral part of the ROV system and outputs of the vision system are fed into to the ROV control loops. The first type of experiments are based on realistic simulation and the second on a real live experiment done with our underwater ROV.

This chapter is organized as follows. The next section, "Related Work" will cover different aspects of localization done underwater including acoustic and vision methods. Section 8.3 will give an overview of the proposed algorithm as well the materials such as the ROV hardware and the simulation used for the research. Section 8.4 will summarize the results we obtained from simulation and real experiment. Section 8.5 will discuss the different key achievements of this work together with our view of future applications.

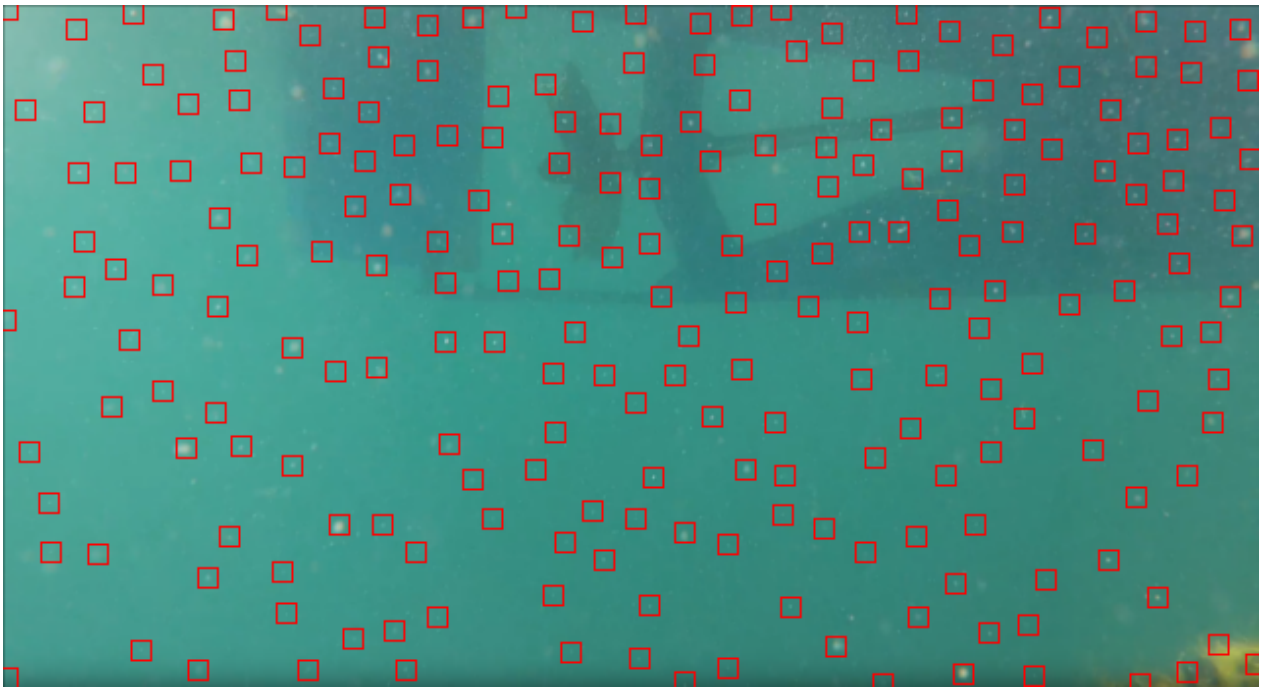


Figure 8.1: This image was taken by lowering our Go-Pro camera in Pelorus Sound, New Zealand. We applied feature extraction algorithm using Shi-Tomasi corner detector (Good Features To Track) implemented in OpenCV [131]. Most of the features in this scenario are detected on plankton particles moving around the ship hull.

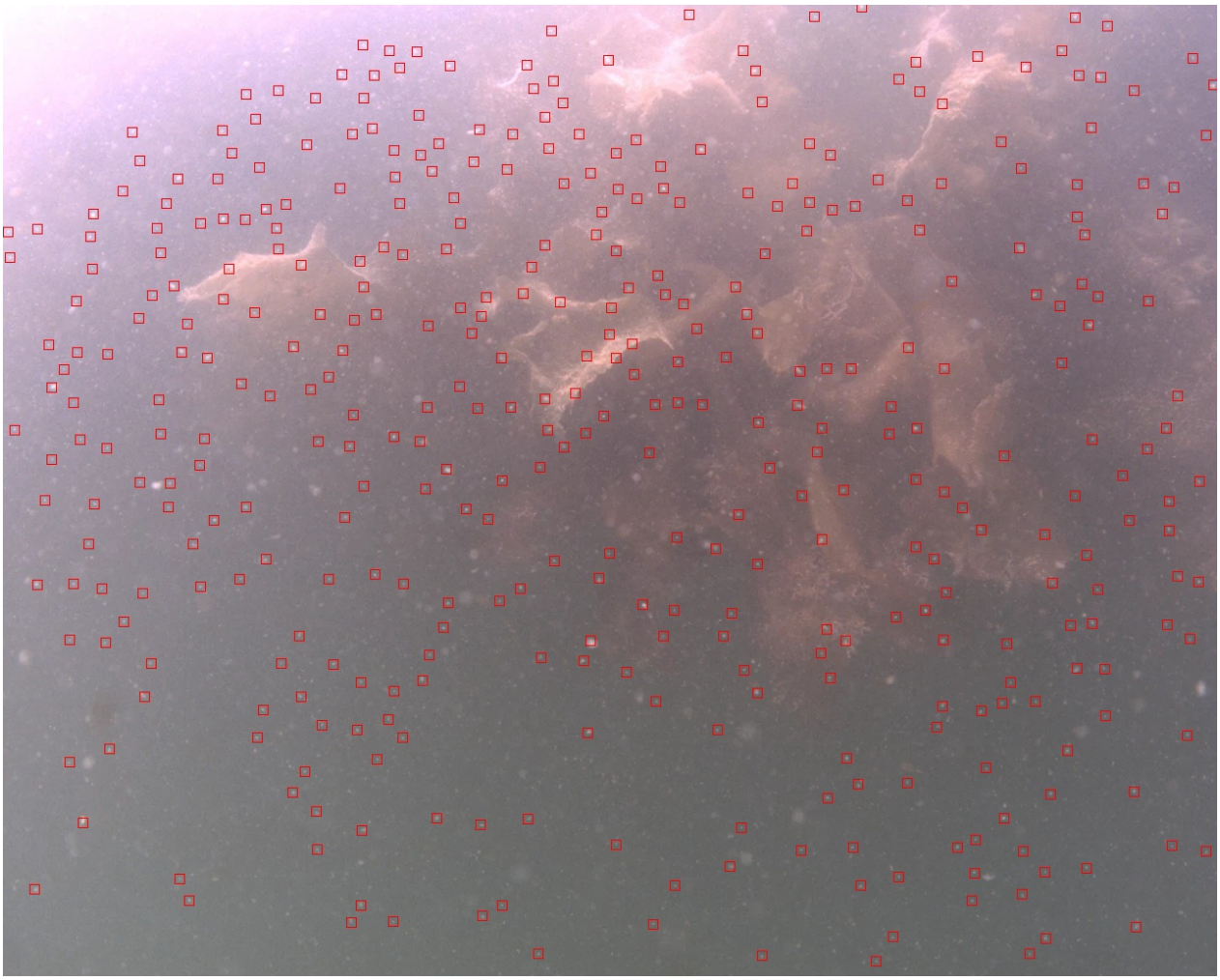


Figure 8.2: This image was taken by our underwater ROV in the Te Ana Marina, Lyttelton, New Zealand. We applied feature extraction algorithm using Shi-Tomasi corner detector (Same as Figure 8.1). Most of the features are detected on the organic matter surrounding the seaweed and not on the seaweed itself.

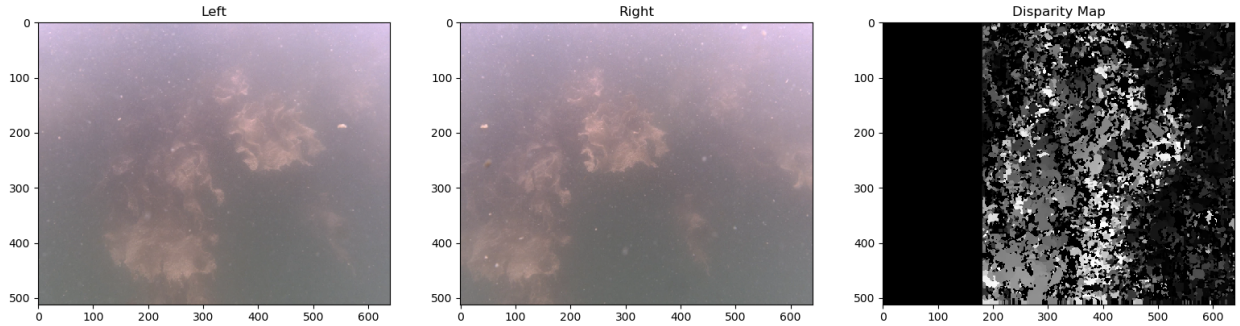


Figure 8.3: A sample stereo pair from the sequence of images obtained from the vision system and the computed disparity map. The images were rectified before running the algorithm. We can see that the disparity map is noisy and practically unusable [130] [132] [133].

8.2 Related Work

The most common approach for underwater localization is using underwater acoustic sensors [134] [135]. In some cases we also take into consideration the movement of the receiver due to surface waves [136]. These are expensive systems and requires positioning an array of receivers

on the surface. The acoustic systems accuracy is decreased as the distance is increased. Also, it is sensitive to multipath interference.

In the computer vision domain, Croke [135] compared the GPS approach to stereo feature based approach, and according to their experiment they got acceptable results using visual odometery. Their work was conducted in a coral reef environment where visibility was good and features could be detected using Harris corner detector and Normalized Cross Correlation (NCC) to match between images. The experiment was done in a clear lake. Our method uses a similar principle but is more robust to the visibility conditions and is more efficient since we only use one correlation window as explained in the following sections.

Paull [137] did a review of the existing technologies for ROV navigation including vision based (SLAM), acoustic methods and Inertial measurement unit (IMU) based methods. In his review he covers the advancement in SLAM research done since previous reviews. SLAM can be classified as feature based methods and the disadvantages of these methods for underwater conditions are demonstrated in figures 8.1 and 8.2.

A more recent review given by Mai [138] concluded that semi-autonomous upgrades to existing ROV technology is a logical step towards the adaption of AUVs for the task of sub-sea infrastructure inspection. Our presented method falls directly into this category as a "station keeping method" aiding in the inspection process.

Aulinas [139] suggested an interesting approach in order to mitigate the underwater feature extraction challenge by segmenting interesting parts in the image using edge detector and applying feature detection only to those regions. This approach reduces the possible matches between frames. This is an approach which we can adopt in our method which is currently based on pre-defined fixed point to specify a region of interest (ROI).

In other domain platforms like drones, 3D stabilizing is usually done using a GPS or optical flow sensor integrated with acoustic range sensor [140]. The stabilization in drones is done with respect to ground. The underwater scenario is more complicated and this kind of setup might be problematic. The objects that we are looking at especially if they are in front of the ROV can have different shapes and might also be moving. This will require adjusting the range of the acoustic sensor beam to the object of interest and cut the relevant ROI from the imaging sensor which will complicate the system even more. A stereo setup in the underwater case makes more sense especially if it already exists as part of the ROV imaging system, where optical flow sensors are usually a stand alone system attached to the bottom of the drone [140].

8.3 Methods And Materials

8.3.1 Template Matching

In our approach for localization we used Normalized Cross Correlation (NCC) which we found suitable for our under water conditions [141]. The NCC is defined in eq. 8.1 to 8.4.

First we used mean subtraction of values from template T. I is also mean-subtracted by moving a sliding window and subtracting the average of that window from each pixel in the I image.

$$T' = T - \frac{1}{(w \cdot h)} \sum_{u,v} T(u, v) \tag{8.1}$$

$$I'(x, y) = I(x, y) - \frac{1}{(w \cdot h)} \sum_{u,v} I(x + u, y + v) \tag{8.2}$$

Where:

$$0 \leq u < w \ , \ 0 \leq v < h \tag{8.3}$$

and w and h are the width and height of the template respectively.

$$R(x,y) = \frac{\sum_{u,v}[T'(u,v) \cdot I'(x+u, y+v)]}{\sqrt{\sum_{u,v} T'^2(u,v) \cdot \sum_{u,v} I'^2(x+u, y+v)}} \tag{8.4}$$

The global maxima value of R defines the matching points of our algorithm. We used the normalized form of the template matching since our stereo setup included two separate time synchronized cameras and each camera had its own gain and exposure control also due to slightly different position of each cameras reflections and refraction effects the luminance of the objects. For simplicity we used the NCC for both sequence matching and the stereo matching but we expect less luminance sensitivity between frames and future improvement can be based on using only Cross Correlation instead of NCC for sequence matching. Although NCC is not ideal for feature tracking due to the fact that it is sensitive to perspective distortions, in a stable ROV setup which maintain a certain position and orientation this sensitivity becomes an advantage over other more invariant methods.

8.3.2 Algorithm Overview

Figure 8.6 shows how we utilize the stereo vision to achieve reliable measurements for the lateral control loop (side to side movement in front of the object) and the range control loop. The images are sampled (synchronized by hardware trigger) and the left image is fed in to Sequence Matching Tracker (SMT). In case of a lock event (given by the operator) the SMT saves a template from the center of the image representing the area of interest. This is very similar to the keyframe concept in many other tracking algorithms. In our case, since we are constantly matching the current image to the template when the locking event is created, no drifting of the translation is expected in contrast to matching only to the previous frame. The stereo matching process is done for each synchronized frames. We take a stereo template from the current left image at the same tracked position as the current sequence matched template and running NCC along the epipolar line on the right image. Since it is the center of the image we expect (under the assumption of a calibrated camera and rectified images) that the matching will be along the horizontal center-line of the right image. The next step is to apply stereo triangulation method using the center points in both images to calculate the range. Figure 8.4 shows all the NCC patterns we used in two consecutive time frames.

We can see in figure 8.5 that the correlation peak is not unique, and therefore cross validation is being done by taking a search area around the matched stereo point on the right image and running NCC with the correlation pattern around this point. The peak in the NCC should be in the center of the search area (up to a threshold) if the stereo match was correct. A higher ratio between the threshold and the cross validation search area will give more statistically significant validation of the result. A lower ratio will mean a smaller search area compared to the threshold and will be faster to compute.

To select the proper correlation window size we need to consider performance trade offs. In addition the search area of the SMT and the cross validation dictates time complexity of $O(n^2)$ whereas the stereo search is $O(n)$ which means we can use a larger window where as the SMT and the cross validation is more sensitive to increasing search area. We found that for our test cases a 60x60 correlation window for the SMT and the cross validation and 80x80 for the stereo was large enough to average the effects of the noise present on our tested environments.

The cross validated data coming from the SMT and the stereo matcher is fed in the form of linear displacements to an alpha-beta position filter [142]. The lateral position is calculated from the pixel offset and the calculated filtered range. It is important to note here that we are only tracking the center which doesn't provide us orientation information and this method

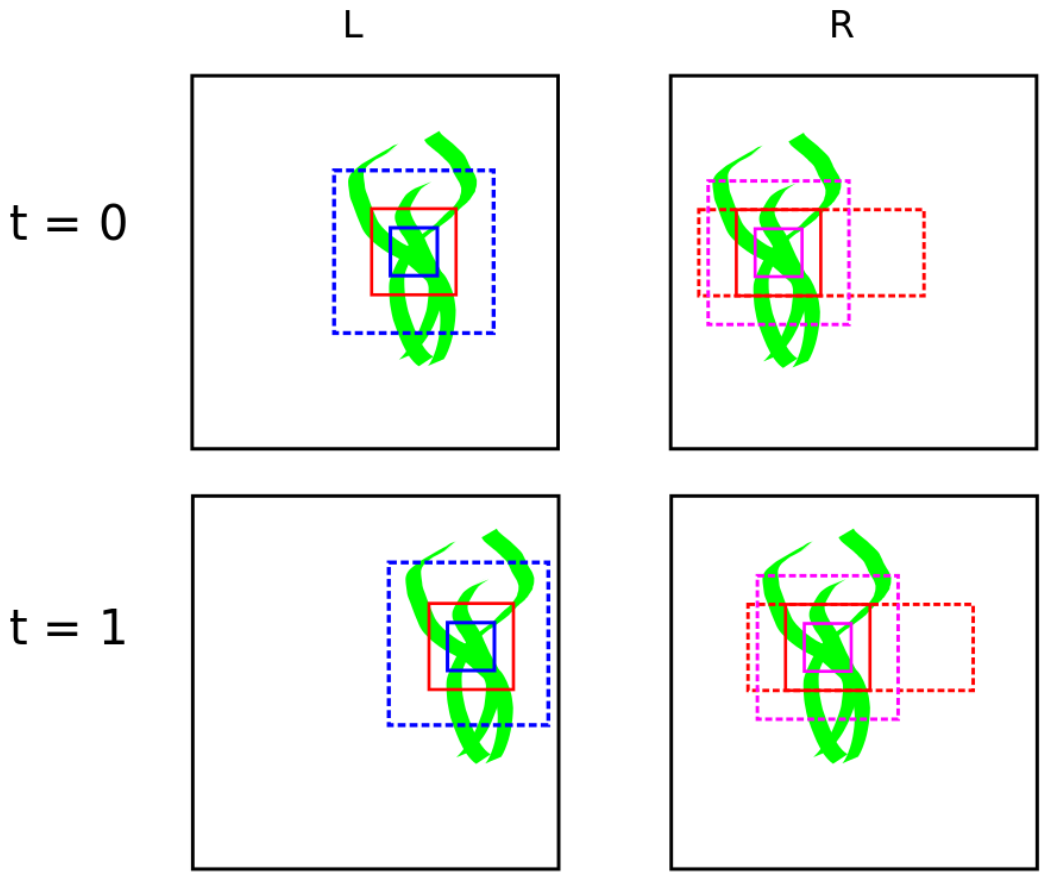


Figure 8.4: NCC patterns used in our algorithm in two different times stamps. The blue line represents the sequence matcher where as the dashed lines represent the search area and the solid line represents the template. In red we can see the template taken from the left image matched to the right image along the horizontal center-line within the search area marked in red dashed line. the magenta pattern shows the cross validation of the template taken from the left image and matched on the right image in the position marked in solid magenta. We can see that the search area windows used for stereo, sequence and cross validation matching are getting updated in the following frame at t=1.

is only valid under the assumption of level platform such as stabilized ROVs. Stabilization in ROV is achieved by having high center of buoyancy and low center of gravity [143]. In a less stable platform we will need to integrate the information from the IMU in order to correctly compensate for the changes in orientation.

Finally, the filtered displacement data is being fed into the Range Control Loop and the Lateral Control Loop which are essentially PID loops controlling the horizontal thrust of the ROV. For completion, although we can get the vertical displacement from the sequence matcher, the vertical control loop is only using the depth meter which is very reliable and accurate (2 mm resolution) and not dependent on the visual characteristics of the surrounding environment [144].

8.3.3 Choosing The Template

Choosing the optimal initial tracking point (the point from which we crop the reference template on the left image) is not trivial since we are using stereo vision and the matching is done not only to the previous frame but also to the right image. The matched stereo position or the matched sequence position from the previous frame does not necessarily exist within the image

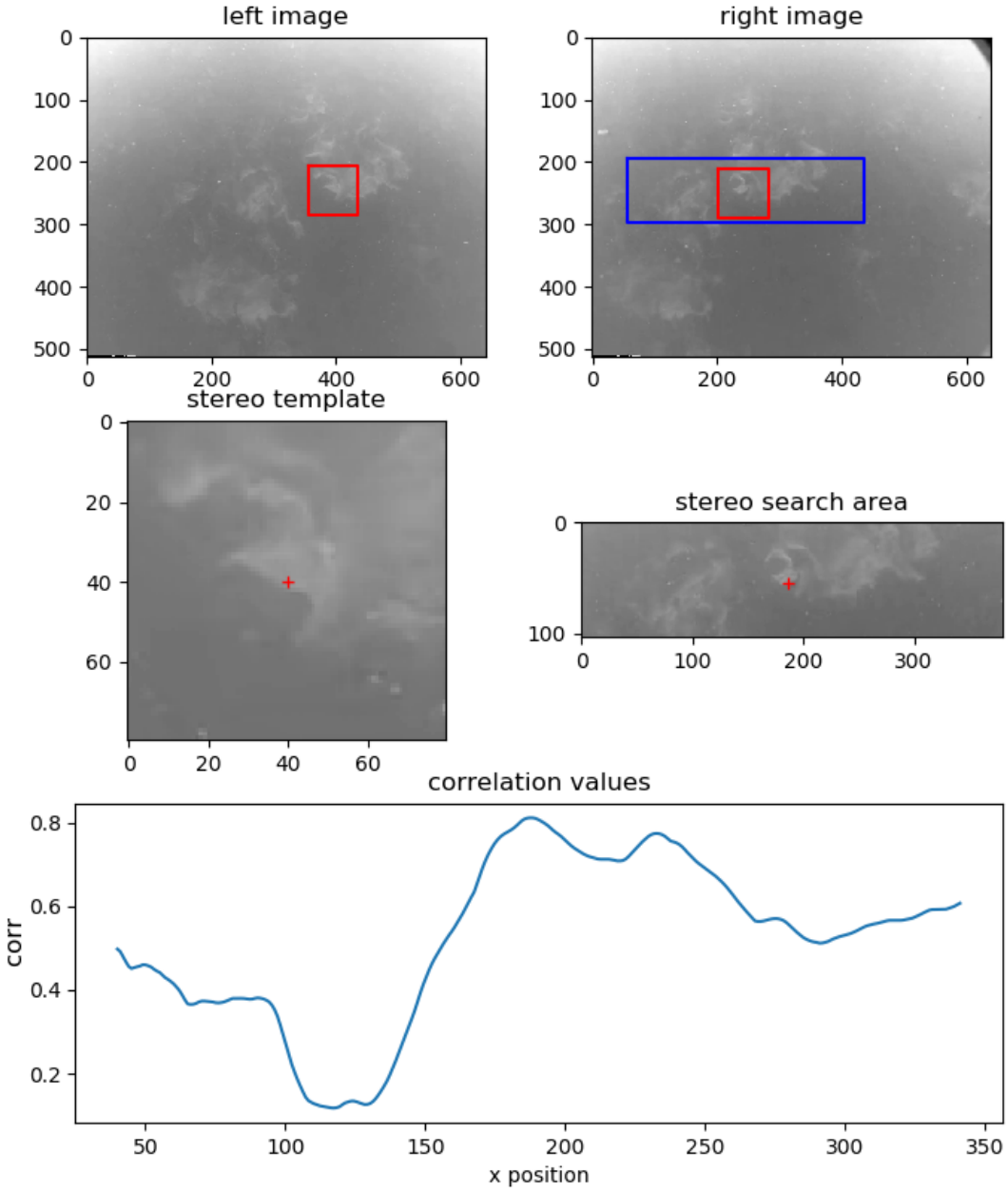


Figure 8.5: The stereo matching process. We can see the template taken from the left image matched to the right image. We choose the peak correlation value as the correct result. We can see the the peak is not unique due to the quality and nature of the the underwater images

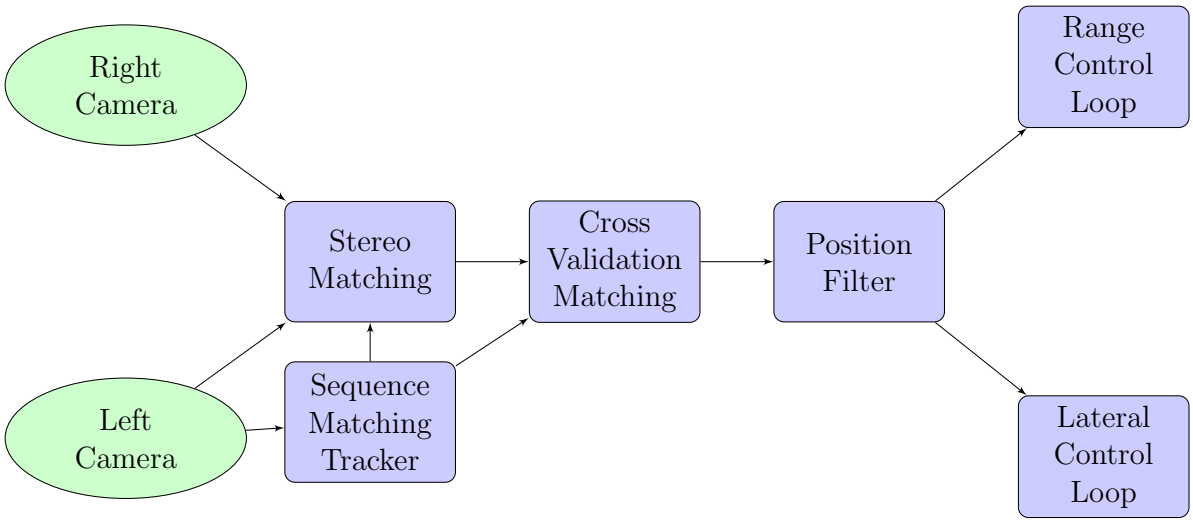


Figure 8.6: Lock event algorithm overview.

boundaries. Our initial approach was to select a point in the center of the left image. The problem with that approach is that for near objects we might not have enough overlap. We also can't choose a point too close to the edge (of the left image) since after tracking a sequence of frames we might go outside the image due to camera movement. Due to this tradeoff and the fact that we are interested in near object (due to visibility limitations) we have chosen to select the initial template position between the center and the edge of the left image depending on a parameter.

8.3.4 Algorithm Variation

A different way of choosing the template is using a feature extractor algorithm in the area (parameter dependent) we mentioned above. We assume this will help us avoid choosing accidentally smooth templates. This approach has become our primary approach for current and future experiments

Another variation is skipping the cross validation and letting the subsequent position filter to filter the noise. This might pass low confident matches and will act as outliers noise. We still can reject matching results based on low correlation threshold or/and inconsistencies with previous matching. At this point it is not clear to us whether this approach is better in terms of the overall performance but since we skip the cross validation we reduce the time needed for this variant.

8.3.5 Simulation

In order to test the concept in a high fidelity manner, we used the DroneSimLab framework which includes a realistic underwater environment [40]. The simulation enabled us to develop the algorithm and to test different strategies suitable for underwater experiments. The simulation was used up to the point we were ready to test it in a real underwater environment. This type of simulation enables us to test algorithms in-context in a similar way to real experiment. This is an important concept since vision algorithms affect the ROV behaviour system wide, and the performance of the algorithm should be considered in the context of next steps in the control pipeline. For example, in the complete system, we might prefer less accurate and more stable algorithms. This delicate trade-off can be tested in-context if a closed loop simulation is used.

Figure 8.7: Our stereo vision ROV. This image shows our ROV before diving in the Lyttelton Te Ana marina (New Zealand) [149]. This a BlueROV [145] type of ROV with a stereo rig we designed for our research. We can see the two camera tubes on the bottom of the ROV

8.3.6 Hardware And Software

The hardware we used for the real experiments was a BlueROV [145] an open platform we equipped with a specially designed rig with two hardware synchronized FLIR cameras (model BFS-U3-13Y3C-C) as can be seen in figure 8.7. Full details on the design of the rig can be found in [146] and the software design docs and code is maintained under GitHub [147] [49]. This is a small size low cost equipment (under 10K USD) for underwater inspection which makes it a perfect candidate for computer vision algorithms to be used to compensate for relatively low quality navigational aids such as IMUs.

The hardware used to run the algorithm was running on our onboard UpSquared ROV computer equipped with the Apollo Lake SoC Pentium N4200 [148]. It is the same processor used in some low-mid range Intel based laptops.

Calibration of the cameras for the purpose of this experiment was done using the stereoCalibrate function from the OpenCV library [27]. The calibration was done under water to take into account the changes in refraction of the camera and the camera port underwater.

8.4 Results

Figure 8.8 shows our simulated ROV equipped with a stereo camera in front of a seaweed. This is (as expected) a challenging environment. The seaweed is constantly moving and simulated plankton (the white spots) covering the scene. In addition, on the edges of the frame a dirt pattern can be seen moving with the camera rig. The attached video [150] shows the unique dynamics of this underwater scene and the successfully close loop simulation where the ROV

Figure 8.8: Simulating tracking experiment using our simulated stereo rig. The inner green square on the left image represents the window we are tracking between consecutive frames the outer square represents the the window we are matching to the right image to extract the range to our target.

is tracking a selected point on the seaweed.

Figure 8.9 shows the overall behaviour of the range control. We can see that the ROV is maintaining the range to the seaweed even though the seaweed is constantly moving and changing shape. We can observe the noise created by the movement of the seaweed effecting the measured range. Overall the range to the seaweed is maintained in this simulated scenario.

In video [151], we can see a result of our real-world experiment of locking on a seaweed in the Te Ana marina captured in a 70 sec video. This was a challenging scenario. The seaweed was moving approximately 20 cm to each direction. We observed tidal currents of approximately 20cm/sec and the visibility was below 60 cm with a lot of organic mater floating around the seaweed. In this video we can see successful closed loop scenario where the ROV stays in front of the seaweed under extreme low visibility conditions and highly dynamic scene. Figure 8.12 shows a snapshot from that experiment with explanation regarding the markings on the frames. Figure 8.10 shows the overall behaviour of the range control loop. The ROV is managed to maintain a range of half a meter from the moving seaweed. Figure 8.11 shows the behaviour of the lateral control loop. We can see that the integral component ('i') is "reacting" to large values in position ('p') induced by the currents pushing the ROV. We can also observe that on frame 3940 the SMT match failed and a new track point was reselected and the ROV continued to maintain its lateral position. This experiment was done with a slightly earlier version of the algorithm where the position of the stereo template was fixed without the cross validation variant mentioned in section 8.3.4.

8.5 Conclusions And Future Research

In this chapter we proposed a novel approach of adopting traditional tracking techniques in a highly visually challenging environment. This approach was demonstrated to be successful in simulation and in real experiments. Furthermore we tested the proposed algorithms in the context of a full system architecture including the control loops. Closed loop realistic simulation was used prior to the real experiments to validate or disprove different methods. We expect this type of simulation to become a sandbox for comparing different algorithms in-context.

We presented in this research a successful localization of an ROV in front of a seaweed

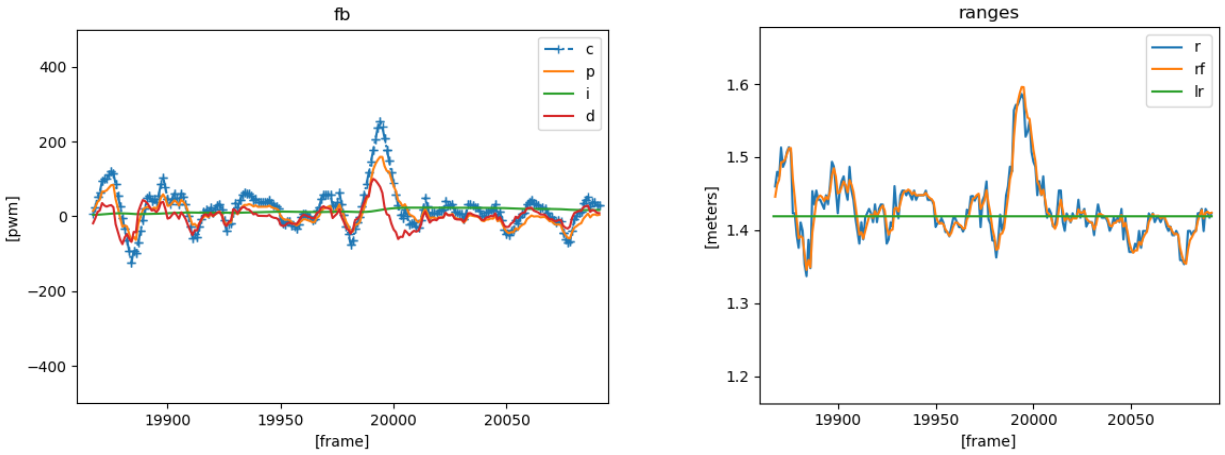


Figure 8.9: Range control. The left graph shows the performance of the PID loop (marked as 'p' for position,'i' for integral, and 'd' for derivative). The blue line (marked c in the legend) represents the command sent to the ROV thruster. We can see that the overall command is oscillating around zero. On the right we can see the actual measured range where 'r' marks the measured range, 'rf' marks the the filtered range mentioned above and 'lr' is the "Lock Range" which is the desired range.

during inspection of a wharf pylon. The ROV was stable in spite of strong tide currents and enabled the operator to easily focus on the inspected seaweed.

Further improvements can be added in the correlation domain and we can look at multiple local maxima as can be seen in figure 8.5. We can for example try different hypothesis for each maxima. In addition as mentioned in 8.3.4 there are a lot of possible variants to this algorithms and a lot of trade-offs. We will continue to test those trades-offs and continue to conduct experiments under those challenging conditions.

We could see that there is a tight connection between the vision algorithms and the control loops. This type of tight coupling can be tested under our closed loop simulation. In real experiments we can usually objectively test only one approach since underwater conditions are changing constantly. Having said that building a data base for open loop type analysis is also important and we will continue collecting data from those environments in the future. This is a parallel type of research in which the algorithms and the overall performance of the ROV directly influence the quality of the data gathering.

To expand the algorithm in terms of following a path or tracking a rope in a more traditional visual odometery, we will need to replace the correlation window each time the confidence level is below a threshold or the cross validation fails. From the triangulation of the new point we can calculate the new camera position. Since we only track one point this can only work with additional input from the IMU that will calculate the orientation.

In terms of run-time performance this algorithm is very efficient and we were able to run this algorithm on a up-squared board [148] onboard the ROV. We tested the algorithm on a modern CPU i7-7700K and it took an average of 4 ms per frame on a single core. We expect those type of light weight algorithms to be more dominant in the underwater domain in the near future where space and heat management are a significant design factors.

Figure 8.10: Range control in real experiment. The markings are the same as figure 8.9.

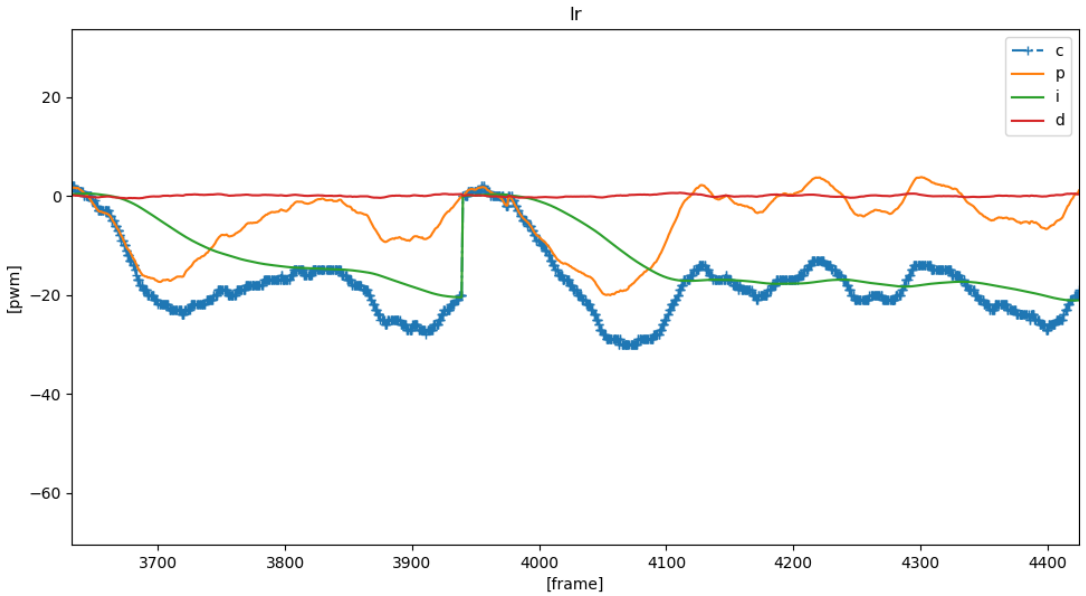


Figure 8.11: PID graph of the lateral control loop. The markings are the same as 8.9.

Figure 8.12: Live experiment done in the Te Ana marina [149]. We can see on the left image the red square window is matched on to the right window, The blue square represents the original position when the lock event occurred and the magenta square is the current matched position.

Chapter 9

Conclusions and Future Research - ROV Simulation Domain

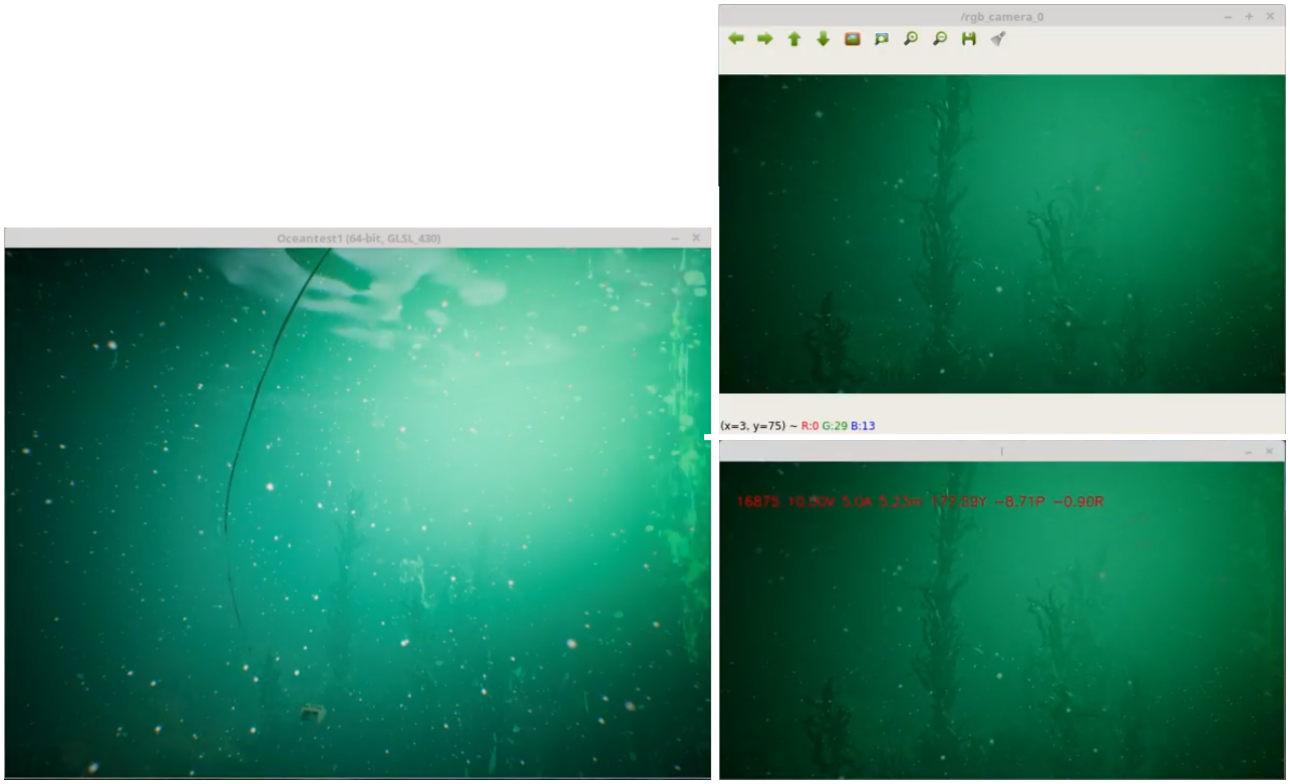


Figure 9.1: Final simulation result. Running the simulation opens 3 windows. The left window shows an outside view of the simulated ROV including the tether. The images on the right are the images from the simulated camera attached to the ROV (one image is with annotations and the other without) The ROV is controlled using an outside gamepad.

Although the environment and the visual effects of a state of the art game engine are highly convincing, some complex and important effects were neglected for the sake of simplicity. Similar to ground effects experienced by airplanes, we have ground underwater effects both in terms of hydrodynamics when getting close to the constraining floor and also the visual effects like small sand particles moving due to the flow created by the ROV. We hope that by showing the usefulness of this work we will encourage improvements and generation of environments not only for the purpose of gaming but for computer vision research in general, and underwater simulation in particular.

The motivation behind our simulation was effectively demonstrated using a popular underwater ROV which we tested in the real-world and used as a reference to our simulation. We

applied computer vision algorithms to test the hypothesis that the created simulated environment and the real environment are compatible to some extent. The computer vision algorithms we test were just a sample and it will be interesting to see more algorithms tested under this environment. Due to the high fidelity nature of the simulation, the simulation can serve as a good baseline for comparison between techniques and approaches in computer vision, control algorithms and the combination of both.

Developing an underwater simulation is a multidisciplinary task involving different expertise from different domains. The existing tools today from the different domains can provide the necessary components. In this part, we covered all the parts which comprise a full-scale simulation. Each part was accompanied by a relevant sub simulation for additional per case tunning. A code reference was provided for reproducibility of the published results. Video [152] and figure 9.1 shows the final result of the simulation which includes the ROV dynamics, the tether simulation and the surrounding underwater environment. The generated simulation framework later used to do a full end to end localization experiment for localizing an ROV infront of an underwater seaweed. We successfully repeated the experiment in the real world and validated the algorithm developed in the simulation.

Our future research will focus on building and comparing more complex underwater ROVs with multiple cameras and ROVs with better manoeuvrability compared to the OpenROV and the BlueROV. Due to the nature of the underwater environment it will also be wise to add additional sensors such as sonar to compensate for the low visibility underwater and use sensor fusion techniques for navigation. To simulate the sonar we can use depth maps provided by the game engine as seen in figure 1.2 and model the necessary noise. We are also aiming for the comparison of a high level algorithms which incorporates computer vision together with control algorithms in both simulated and real environments. For example, in our simulation, an autonomous mission can be tested and analyzed end to end before applying or even building an ROV. The algorithm presented in chapter 8 was the first high level algorithm we developed and is a first step towards that direction. We can use the dynamic section presented in chapter 7 to guide us in designing the ROV and also manipulating the underwater environment we presented for the purpose of that mission. A fully autonomous control algorithm like a seabed scan can be designed and tested under the presented framework.

Part IV

Future Research And Conclusions

Chapter 10

Future and Ongoing Research

10.1 Buoy Camera Simulation

In previous parts, we discussed the drone simulation domain and the underwater simulation domain. In this section, we present our current work (ongoing) on a practical test case of using game engine based simulation. This work is done with the aim of providing a future user the ability to watch/analyze the video coming from a camera mounted on a buoy with a pole about 1 meter above the surface. The camera is used to monitor the status of sea structure like offshore fish farm. After creating a successful simulation of the camera mounted on the pole, a demand aroused to stabilize the video before presenting it to a potential viewer. We used the game engine simulation framework to test a stabilization method based on the video stream and IMU data coming from the UE4. To simulate the sea surface, we used an existing asset of physical water surface downloaded from the UE4 marketplace [153]. The simulated IMU provides Euler angles from which we took the roll and the pitch angles to stabilize the generated video.

At this stage of the project it is unclear whether the real mounted camera will include an IMU or whether the IMU will be synchronized to the camera and the simulation was presented as a future tool to be used for evaluating either IMU based stabilization or computer vision based stabilization [154].

The stabilisation process is straight forward since we are using a perfect sensor under the simulation. First, we compensate for the pitch and calculate the pixel translation in the vertical axis as can be seen in the following equation:

$$dy = 0.5R\frac{tan(\theta)}{tan(1/2\phi)} \tag{10.1}$$

Where dy is the offset in pixels in the vertical direction, R is the number of rows, θ is the sensor pitch angle and ϕ is the vertical FOV of our camera. The final Affine transformation can be obtained by multiplying a 2D translation matrix and a 3x3 rotation matrix as described below

$$T = \begin{bmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{bmatrix} \tag{10.2}$$

Where θ is the roll angle and T is the final image transformation. At this point, we only take into account the effects of the roll and the pitch and not the camera movement. This is important since getting the pitch and the roll is a relatively easy task and can be obtained by using a low-cost tilt sensor. Getting the relative height change is not straight forward and requires sensor fusion approach in order to use simple low-cost MEMS sensor such as accelerometers or pressure sensors.

Figure 10.1 shows the rotation angles arriving from the simulation. These angles are simulating a perfectly noise free and synchronized sensor. We can see the rotation of the camera between -10 to 10 degrees in roll and pitch, and less than 1 degree in the yaw angle as we would expect due to the wave movement. In Figure 10.3 we can see the camera motion in the 3D space. We can see that the camera is moving in all direction and not only in z due to the fact that it is mounted on a pole. Video [155] shows the original simulation video stream side by side with the stabilized one. Figure 10.3 shows single frame before and after Roll and Pitch stabilisation.

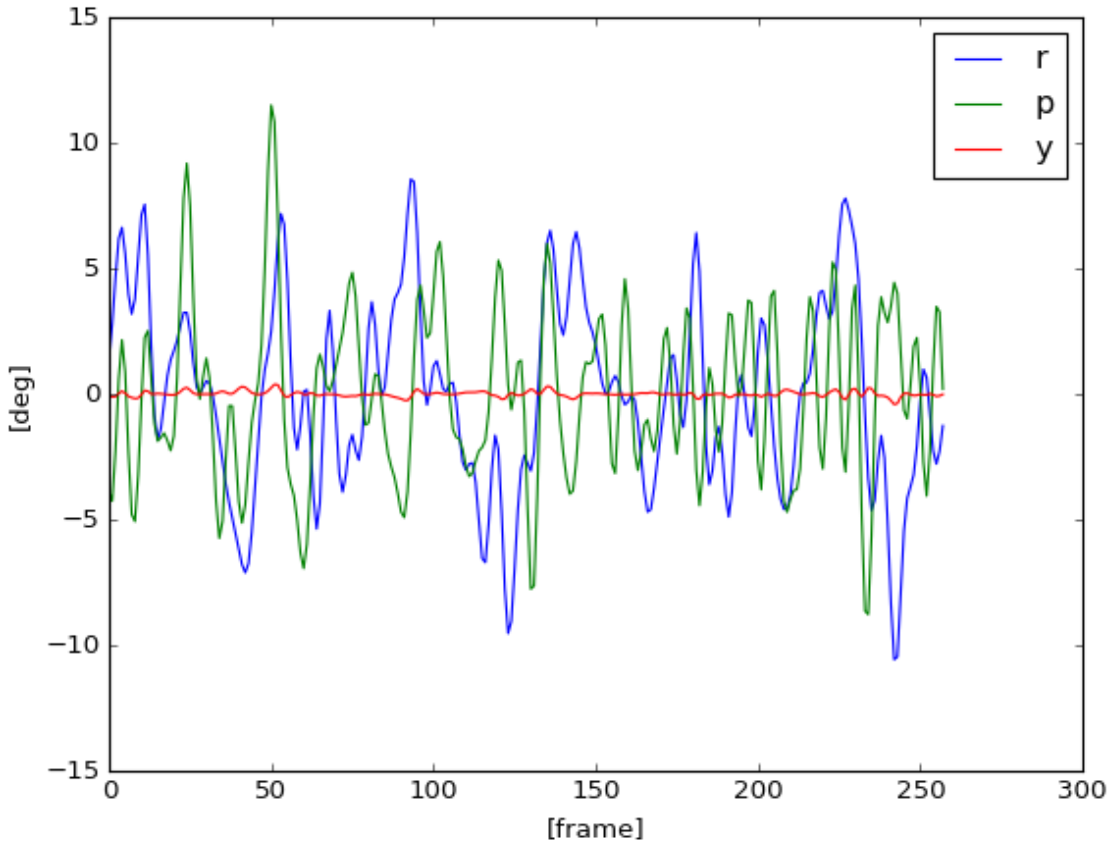


Figure 10.1: Image stabilization experiment input Euler Angles.

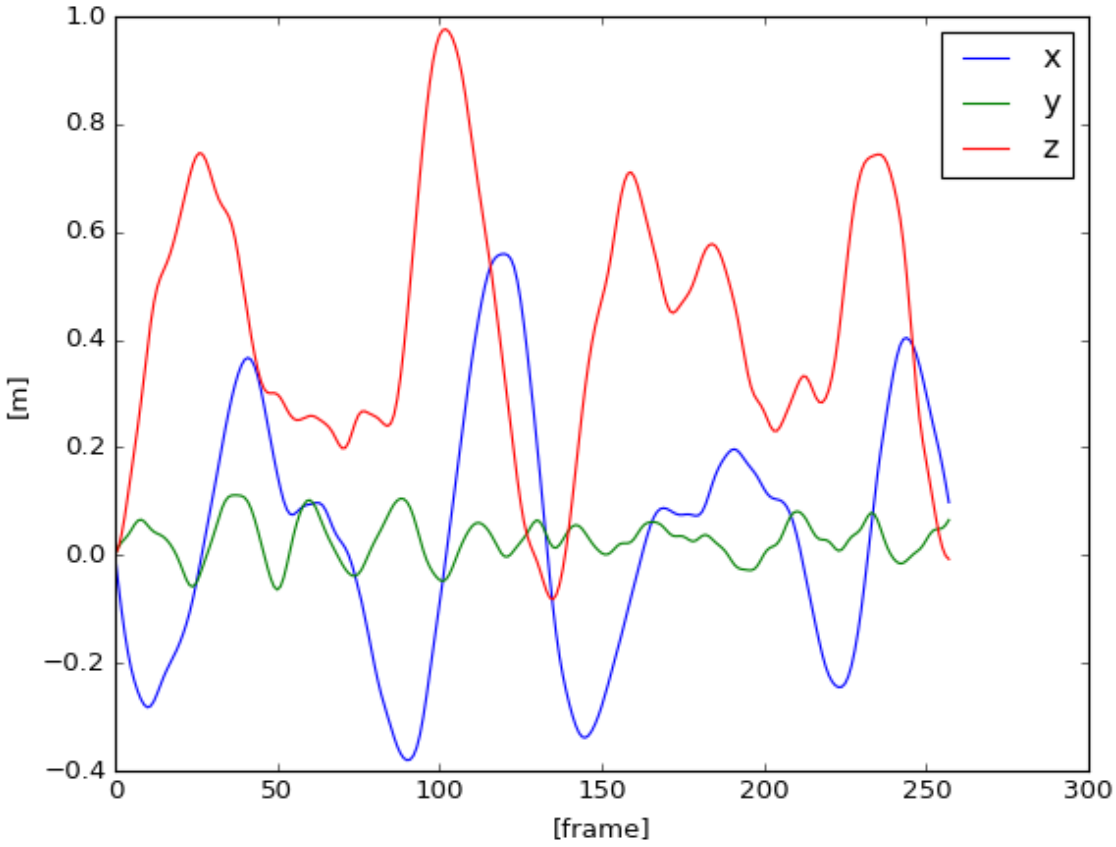


Figure 10.2: Image stabilization experiment camera position.

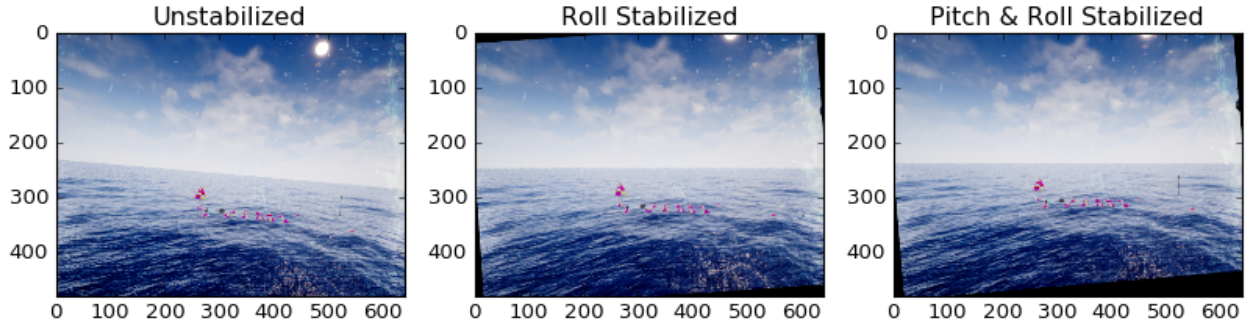


Figure 10.3: Image stabilization before and after. From left to right we can see the unstabilized image , The image after only roll stabilization and the image after stabilizing for roll and pitch.

Although we haven't applied computer vision methods for this domain as we did for others, it clearly shows the visual effects that are expected to challenge any upcoming algorithms. We can see that the contribution of the game engine is significant. The waves and ripples created by the sea surface, the lights, the atmospheric conditions and the dirt mask on the camera port will all affect future vision algorithms applied to this scene.

10.2 Underwater Visual Odometry

The research described in this thesis opened up an exciting new way for developing computer vision algorithms. In our research, we presented the usefulness of large correlation window for

localisation. We are now currently expanding this algorithm to general Visual Odemetry (VO). Figure 10.4 shows a snapshot taken in data collection mission in Golden Bay New Zealand. In this mission, we followed a rope to maintain orientation. This manoeuvre is common for divers in scanning seabed missions.

As explained in chapter 8, traditional SLAM methods are not reliable in real underwater environments and since large correlation window based tracking showed promising results we are using it also for VO. Figure 10.5 shows our simulation setup and initial results of following a line. Figure 10.6 shows a successful experiment in a swimming pool. This technique will help us in future dives by removing the need for a rope as seen in figure 10.4.

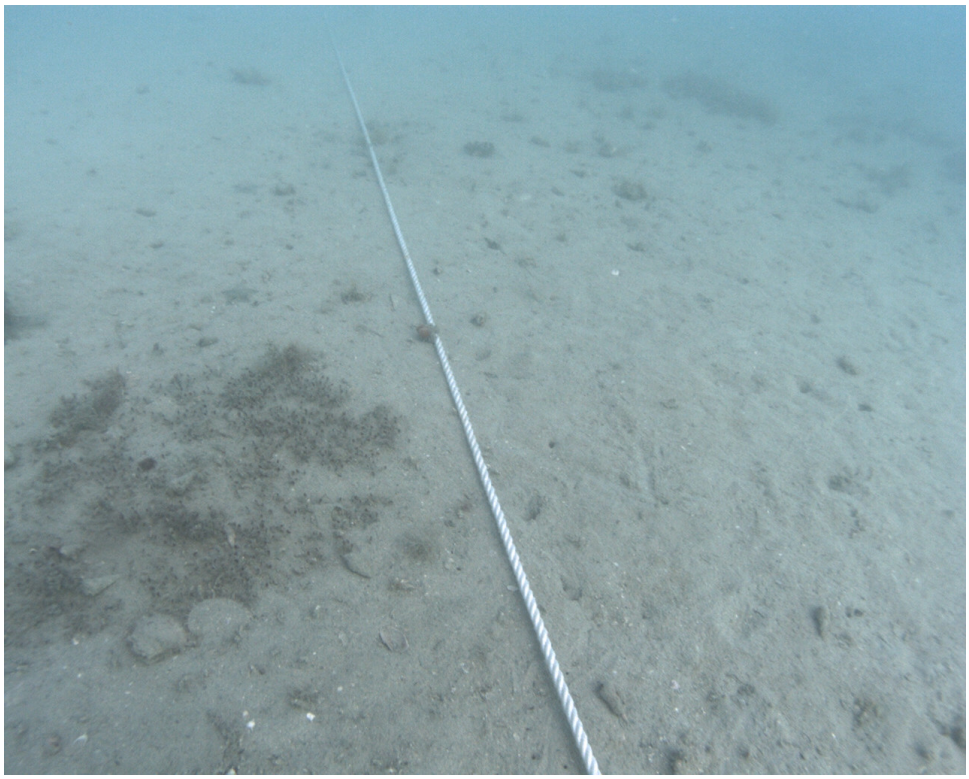


Figure 10.4: An image taken by our ROV in a data collection mission in Golden Bay New Zealand.

In figure 10.5 we can see an experiment of our simulated stereo ROV following a faint line on the sandy ground. In the future, we will increase the complexity of this scene to get more realistic influence on our algorithm.

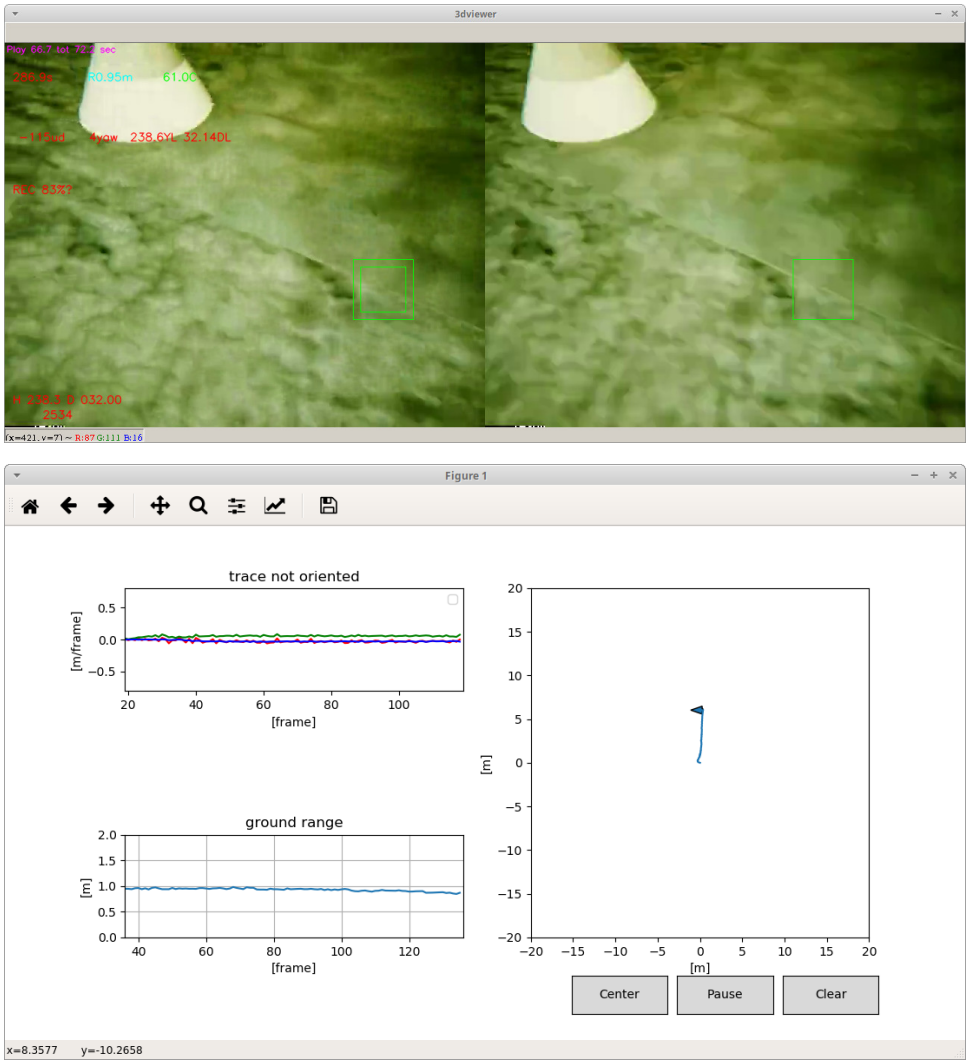


Figure 10.5: A simulated experiment of following a line on a seabed sandy environment. The top image is a snapshot from the simulation whereas the lower image shows the trace on an X-Y plot where the orientation of the ROV is represented by a small arrow. The range of the ROV to the ground and the calculated trace between the frames is also given. These plots are also given in realtime to assist the pilot in orientation. In the image, we can see a cone we placed to mark the beginning of the line.

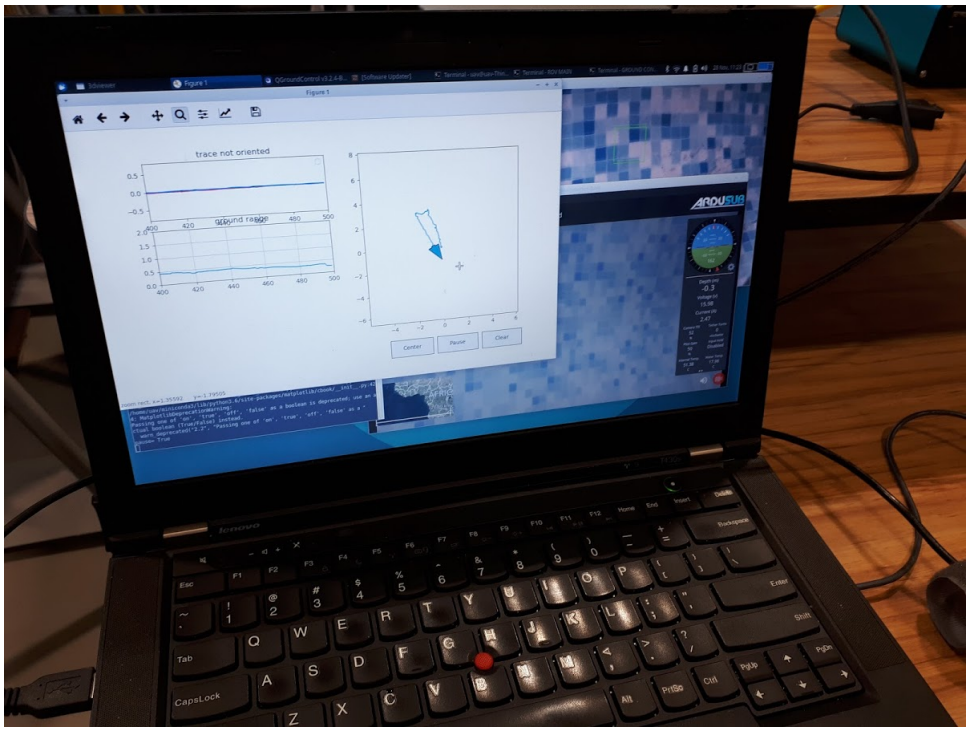


Figure 10.6: Visual odometry pool experiment. A picture taken during an experiment of performing a U shape maneuver. The U pattern is shown in realtime during the experiment.

10.3 Future Plans and Activities

In addition to the ongoing research, we are also working in the mechanical domain to improve our simulation in terms of more accurately estimate the ROV dynamics. For example, currently, we don't have an accurate model of the thruster. More accurate thruster model will help better design the control loops in our simulation.

The visual domain also needs strengthening. When the ROV touches the ground or even moves close to the ground, a cloud of sand or dirt rises. We can simulate that effect using the particle engine implemented in UE4.

As explained in section 1.3.1, the RTX technology is going to change the quality and the realism of the visual simulation. We expect that this technology will add additional strength to our simulation above and below water. Future research will integrate the new technology in our simulations depending on the RTX support by the game engines.

Another important and needed future research is exporting and collaborating existing simulations and environments. Currently, in the DroneSimLab we are providing the two environments (the underwater and the forest) as part of the package. A more collaborative setup is needed in order for researchers to be able to publish their own environments and robots.

It is clear that this is a growing exciting field and is indirectly supported by the gaming industry. By looking at the rate of the increased realism in today's games we predict that the usefulness of the game engine for simulation purposes will increase accordingly.

Chapter 11

Conclusions

This research started with the hypothesis that using game engine based simulation can increase the productive process of not only system development as already known by developers but also the productiveness of algorithm development. Up until now, a more traditional approach was taken where an algorithm was tested on a real recorded image database such as KITTI [17] or The Oxford RobotCar dataset [16]. The emerging new scientific simulators as described in 1.3.2 are now providing a new end to end benchmark approach.

In our research, we surveyed the latest state of the art game engines (Appendix B) and picked the Unreal Engine 4 to be our main tool to test computer vision concepts. We continued developing a research tool that will connect real vision software systems with the UE4 (Chapter 2). This tool is constantly maintained and used to communicate with new virtual environments we created along the research.

As a first test case scenario, we created a simulated natural environment and a feature tracking algorithm for a moving camera. We demonstrated the tracker performance under different environmental conditions. We also presented a more complex scenario involving a drone tracking another drone entering a forest-like environment.

As a logical next step, we took a computer vision practical problem. We proposed a bounding technique from the sensor fusion domain and validated the results published in that paper using our realistic simulation (Chapter 3). The validation was done by comparing the simulation results to the results from our real world experiment (Chapter 3.5).

The natural environment which included moving trees and shadows emphasised the value of using game engines for recreating real problems that need to be considered when working in natural environment. Existing scientific oriented simulations are not realistic enough for that purpose and other simulation frameworks are more focused on the urban space (Section 1.4). The game engine is designed for giving the user the ability to create realistic environments based on his requirements and existing assets.

After the drone simulation domain, our main focus was shifted to the underwater world. The simulation we developed included sea objects like rocks, fish, seaweed, bubbles and more. In order to reach the desired effect we enriched the environment with plankton models (Chapter 6). The plankton models based on particle filters clearly showed the effects on the visibility in some underwater conditions. This ability to control the conditions reflects the high contribution of the game engines compared to relying on real recorded data. Adapting to variable conditions is one of the reasons why heavily AI based systems such as autonomous cars are still struggling to find solutions in harsh conditions such as snow and fog due to the lack of training data. Our ability to control the environment shaped the computer vision algorithms we used later on in the underwater domain. In some of our dive tests, we discovered that visually simulating the tether is important in cases where the tether is getting in the field of view of the ROV cameras. The tether simulation was added as part of the ROV simulation

(Chapter 5). In addition, we investigated the dynamics of an ROV and suggested a general step by step process for building a dynamic simulation of an ROV based on Kane's method. The simulated underwater environment together with the dynamic simulation was published as generalised ROV underwater simulation (Chapter 7).

The strength of this realistic simulation was demonstrated in our latest work which involved solving a localisation problem in an extremely challenging underwater environment (Chapter 8). We used the simulation to reject the existing computer vision solutions and to validate a new solution which is more suitable for that environment. The decision to reject existing computer vision solutions was later validated on post analysis of the experiment recorded underwater videos. We, later on, tested the algorithm in a real underwater environment using a real ROV. The algorithm was tested in-context with the control algorithms which means it effected in real-time the ROV manoeuvres. This is a major contribution of the game engine that allowed us to test the algorithm and to see its effect on the ROV manoeuvres. In the real experiment, we successfully managed to maintain a constant relative 3D position with respect to constantly moving seaweed. To the best of our knowledge, this is the first time computer vision algorithms were successfully applied to that type of environment.

In addition to the above two simulated environments we also simulated the sea surface, especially the wave movements and the sea dynamics as described in our ongoing research work (Section 10.1). This game engine based simulation is used to evaluate the visual performance of image stabilisation algorithm prior to the installation of a real camera on a sea buoy. This is an ongoing research and is done together with the engineering effort for providing a stable camera user experience on a moving sea surface platform.

Under the computer vision research we continue to suggest new techniques for underwater visual odemetry as described in section 10.2. The game engine framework was used to evaluate the performance of the algorithm and was a predecessor for our latest successful underwater tests.

As seen in our research, game engines have the potential to bridge the gap between the availability of doing experiment in the real world which might be limited and the high demand for data and experiments in the computer vision field. For example, to train an AI model, one needs sometimes hundreds of thousands of annotated samples to cover a wide area of the problem domain. This usually means that researchers are bound to work on popular fields such as autonomous driving where there will be most likely enough data to test their methods. Usually, we don't have enough resources to hire an army of annotators to annotate data from a new domain or even enough resources to gather the data. This is also true for traditional computer vision algorithms such as template-based tracking we discussed earlier. Generally speaking, in the underwater domain gathering enough data to develop a robust algorithm is a challenge. It was especially true in that case (chapter 8), since the quality of the data gathering was highly dependent on the algorithm itself. During the experiments, we didn't need to stabilize the ROV manually and the algorithm help us stabilize the ROV in-front of the seaweed in real-time increasing the quality of the data gathering for further analysis.

Together with those final results, we conclude that game engines although they are meant to be used for the purpose of entertainment, can also be successfully applied to computer vision research. In addition, game engines are widely available and usually free for research and academic purposes. We also conclude that the latest trends and advancements in the GPU hardware especially the real-time ray-tracing capabilities will continue to increase the relevance of those simulations.

Bibliography

[1] Epic Games. Open world. `https://docs.unrealengine.com/latest/INT/Engine/OpenWorldTools/`.

[2] Otoy. Octanerender. https://home.otoy.com/render/octane-render/.

[3] Epic Games. Unreal engine 4 supports microsofts directx raytracing and nvidia rtx. `https://www.unrealengine.com/en-US/blog/unreal-engine-4-supports-microsoft-s-directx-raytracing-and-nvidia-rtx`, 2018.

[4] Daniel Valente De Macedo, Ygor Rebouças Serpa, Maria Andr&eqcute Rodrigues, et al. Fast and realistic reflections using screen space and gpu ray tracinga case study on rigid and deformable body simulations. *Computers in Entertainment (CIE)*, 16(4):5, 2018.

[5] Per Ganestam and Michael Doggett. Real-time multiply recursive reflections and refractions using hybrid rendering. *The Visual Computer*, 31(10):1395–1403, 2015.

[6] Chakravarty R Alla Chaitanya, Anton S Kaplanyan, Christoph Schied, Marco Salvi, Aaron Lefohn, Derek Nowrouzezahrai, and Timo Aila. Interactive reconstruction of monte carlo image sequences using a recurrent denoising autoencoder. *ACM Transactions on Graphics (TOG)*, 36(4):98, 2017.

[7] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and Service Robotics*, 2017.

[8] Abel Gawel, Carlo Del Don, Roland Siegwart, Juan Nieto, and Cesar Cadena. X-view: Graph-based semantic multi-view localization. *IEEE Robotics and Automation Letters*, 3(3):1687–1694, 2018.

[9] Manolis Savva, Angel X. Chang, Alexey Dosovitskiy, Thomas Funkhouser, and Vladlen Koltun. MINOS: Multimodal indoor simulator for navigation in complex environments. *arXiv:1712.03931*, 2017.

[10] Jingwei Zhang, Lei Tai, Peng Yun, Yufeng Xiong, Ming Liu, Joschka Boedecker, and Wolfram Burgard. Vr-goggles for robots: Real-to-sim domain adaptation for visual control. *IEEE Robotics and Automation Letters*, 2019.

[11] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. CARLA: An open urban driving simulator. In *Proceedings of the 1st Annual Conference on Robot Learning*, pages 1–16, 2017.

[12] Joseph DAmbrosio, Arun Adiththan, Edwin Ordoukhanian, Prakash Peranandam, S Ramesh, Azad M Madni, and Padma Sundaram. An mbse approach for development of resilient automated automotive systems. *Systems*, 7(1):1, 2019.

[13] Xue Bin Peng, Glen Berseth, KangKang Yin, and Michiel van de Panne. Deeploco: Dynamic locomotion skills using hierarchical deep reinforcement learning. *ACM Transactions on Graphics (Proc. SIGGRAPH 2017)*, 36(4), 2017.

[14] Nikolaus Mayer, Eddy Ilg, Philipp Fischer, Caner Hazirbas, Daniel Cremers, Alexey Dosovitskiy, and Thomas Brox. What makes good synthetic training data for learning disparity and optical flow estimation? *International Journal of Computer Vision*, pages 1–19, 2018.

[15] NVIDIA DRIVE CONSTELLATION. `https://www.nvidia.com/en-us/self-driving-cars/drive-constellation/`, 2018.

[16] Will Maddern, Geoffrey Pascoe, Chris Linegar, and Paul Newman. 1 year, 1000 km: The oxford robotcar dataset. *The International Journal of Robotics Research*, 36(1):3–15, 2017.

[17] Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for autonomous driving? the kitti vision benchmark suite. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012.

[18] Pedro Serra, Rita Cunha, Tarek Hamel, David Cabecinhas, and Carlos Silvestre. Landing on a moving target using image-based visual servo control. In *53rd IEEE Conference on Decision and Control*, pages 2179–2184. IEEE, 2014.

[19] OSRF. *Gazebo*. `http://gazebosim.org`.

[20] Johannes Meyer, Alexander Sendobry, Stefan Kohlbrecher, Uwe Klingauf, and Oskar Von Stryk. Comprehensive simulation of quadrotor uavs using ros and gazebo. In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, pages 400–411. Springer, 2012.

[21] Ilya Afanasyev, Artur Sagitov, and Evgeni Magid. Ros-based slam for a gazebo-simulated mobile robot in image-based 3d model of indoor environment. In *International Conference on Advanced Concepts for Intelligent Vision Systems*, pages 273–283. Springer, 2015.

[22] Gilberto Echeverria, Nicolas Lassabe, Arnaud Degroote, and Séverin Lemaignan. Modular open robots simulation engine: Morse. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 46–51. IEEE, 2011.

[23] Arnaud Degroote, Pierrick Koch, and Simon Lacroix. Integrating Realistic Simulation Engines within the MORSE Framework. In *Workshop on Rapid and Repeatable Robot Simulation (R4 SIM), at Robotics: Science and Systems*, Roma, Italy, July 2015.

[24] Jon Berndt. Jsbsim: An open source flight dynamics model in c++. In *AIAA Modeling and Simulation Technologies Conference and Exhibit*, page 4923, 2004.

[25] Adam Lerer, Sam Gross, and Rob Fergus. Learning physical intuition of block towers by example. *CoRR*, abs/1603.01312, 2016.

[26] Epic Games. Unreal engine 4. http://www.unrealengine.com.

[27] G. Bradski. The opencv library. *Dr. Dobb's Journal of Software Tools*, 2000.

[28] Weichao Qiu and Alan Yuille. Unrealcv: Connecting computer vision to unreal engine. *arXiv preprint arXiv:1609.01326*, 2016.

[29] Matthias Mueller, Neil Smith, and Bernard Ghanem. A benchmark and simulator for uav tracking. In *European Conference on Computer Vision*, pages 445–461. Springer, 2016.

[30] John Skinner, Sourav Garg, Niko Sünderhauf, Peter Corke, Ben Upcroft, and Michael Milford. High-fidelity simulation for evaluating robotic vision performance. In *Intelligent Robots and Systems (IROS), 2016 IEEE/RSJ International Conference on*, pages 2737–2744. IEEE, 2016.

[31] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. Aerial Informatics and Robotics platform. Technical Report MSR-TR-2017-9, Microsoft Research, 2017.

[32] SCS Software. Euro truck simulator 2. https://eurotrucksimulator2.com/.

[33] Seongjoon Chu Chiwan Song, JaeCheol Sim. Chosuntruck. https://github.com/bethesirius/ChosunTruck.

[34] German Ros, Laura Sellart, Joanna Materzynska, David Vazquez, and Antonio M Lopez. The synthia dataset: A large collection of synthetic images for semantic segmentation of urban scenes. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3234–3243, 2016.

[35] Apostolia Tsirikoglolu, Joel Kronander, Magnus Wrenninge, and Jonas Unger. Procedural modeling and physically based rendering for synthetic data generation in automotive applications. *arXiv preprint arXiv:1710.06270*, 2017.

[36] Ori Ganoni and Ramakrishnan Mukundan. A framework for visually realistic multi-robot simulation in natural environment. *arXiv preprint arXiv:1708.01938*, 2017. WSCG 2017 proceedings.

[37] Ori Ganoni, Ramakrishnan Mukundan, and Richard Green. A multi sensory approach using error bounds for improved visual odometry. In *2017 International Conference on Image and Vision Computing New Zealand (IVCNZ)*, pages 1–6. IEEE, 2017.

[38] Ori Ganoni, Ramakrishnan Mukundan, and Richard Green. Visually realistic plankton models for simulating underwater environments. In *2018 22nd International Conference Information Visualisation (IV)*, pages 420–425. IEEE, 2018.

[39] Ori Ganoni, R Mukundan, and R Green. Visually realistic graphical simulation of underwater cable. In *Proceedings of the 26th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision, Plzen, Czech Republic*, volume 28, 2018.

[40] Ori Ganoni, Ramakrishnan Mukundan, and Richard Green. A generalized simulation framework for tethered remotely operated vehicles in realistic underwater environments. *Drones*, 3(1):1, 2019.

[41] Ganoni G. Ue4pyserver. https://github.com/orig74/UE4PyServer.

[42] Ganoni G. Ue4pybridge. https://github.com/orig74/UE4PyhtonBridge.

[43] Ganoni O. Dronesimlab. https://github.com/orig74/DroneSimLab.

[44] *ardupilot*. `http://ardupilot.com`.

[45] Turbulance pull request. https://github.com/ArduPilot/ardupilot/pull /5205.

[46] Cable sim project. `https://github.com/UnderwaterROV/UWCableComponent`.

[47] Cable sim notebook. `https://github.com/UnderwaterROV/underwaterrov/blob/master/notebooks/rope.ipynb`.

[48] Rov dynamics notebook. `https://github.com/UnderwaterROV/underwaterrov/blob/master/notebooks/openrov_sim.ipynb`, 2018.

[49] Rov software. `https://github.com/uc-csse/RovVision`.

[50] Ganoni G. Ardupilot fork. https://github.com/orig74/ardupilot.

[51] Laminar Research. X-plane. http://www.x-plane.com/.

[52] flightgear. *flightgear*. `http://www.flightgear.org`.

[53] Ganoni G. Px4 firmware fork. https://github.com/orig74/Firmware.

[54] A Babushkin. Jmavsim. https://pixhawk.org/dev/hil/jmavsim, 2018.

[55] GitHub Inc. Github. https://github.com.

[56] Epic Games. Scene capture 2d. https://docs.unrealengine.com/latest/INT/Resources/ContentExamples/Reflections/1_7.

[57] Thomas Mooney. *Unreal Development Kit Game Design Cookbook*. Packt Publishing Ltd, 2012.

[58] Unreal engine 4 with python & opencv. https://youtu.be/q8kAooRaf7g.

[59] Jean-Yves Bouguet. Pyramidal implementation of the affine lucas kanade feature tracker description of the algorithm. *Intel Corporation*, 5(1-10):4, 2001.

[60] Drone tracking drone in dronesimlab. https://youtu.be/Mj9xZECG40Q.

[61] Adrian Rosebrock. Ball tracking with opencv. http://www.pyimagesearch.com/2015/09/14/ball-tracking-with-opencv/.

[62] Raúl Mur-Artal, J. M. M. Montiel, and Juan D. Tardós. ORB-SLAM: a versatile and accurate monocular SLAM system. *IEEE Transactions on Robotics*, 31(5):1147–1163, 2015.

[63] Jakob Engel, Thomas Schöps, and Daniel Cremers. Lsd-slam: Large-scale direct monocular slam. In *European Conference on Computer Vision*, pages 834–849. Springer, 2014.

[64] Michael Bloesch, Sammy Omari, Marco Hutter, and Roland Siegwart. Robust visual inertial odometry using a direct ekf-based approach. In *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*, pages 298–304. IEEE, 2015.

[65] Stefan Leutenegger, Simon Lynen, Michael Bosse, Roland Siegwart, and Paul Furgale. Keyframe-based visual–inertial odometry using nonlinear optimization. *The International Journal of Robotics Research*, 34(3):314–334, 2015.

[66] Timo Hinzmann, Thomas Schneider, Marcin Dymczyk, Andreas Schaffner, Simon Lynen, Roland Siegwart, and Igor Gilitschenski. Monocular visual-inertial slam for fixed-wing uavs using sliding window based nonlinear optimization. In *International Symposium on Visual Computing*, pages 569–581. Springer, 2016.

[67] Richard Hartley and Andrew Zisserman. Multiple view geometry in computer vision second edition. *Cambridge University Press*, 2000.

[68] Zhengyou Zhang. A flexible new technique for camera calibration. *IEEE Transactions on pattern analysis and machine intelligence*, 22(11):1330–1334, 2000.

[69] João Luís Marins, Xiaoping Yun, Eric R Bachmann, Robert B McGhee, and Michael J Zyda. An extended kalman filter for quaternion-based orientation estimation using marg sensors. In *Intelligent Robots and Systems, 2001. Proceedings. 2001 IEEE/RSJ International Conference on*, volume 4, pages 2003–2011. IEEE, 2001.

[70] Scipy optimize. `https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.least_squares.html`.

[71] Ms5611. `http://www.te.com/commerce/DocumentDelivery/DDEController?Action=srchrtrv&DocNm=MS5611-01BA03&DocType=Data+Sheet&DocLang=English`.

[72] Hmc5883l. `https://cdn-shop.adafruit.com/datasheets/HMC5883L_3-Axis_Digital_Compass_IC.pdf`.

[73] Mpu6050. `https://www.invensense.com/wp-content/uploads/2015/02/MPU-6000-Datasheet1.pdf`.

[74] Stm32f103. `http://www.st.com/en/microcontrollers/stm32f1-series.html?querycriteria=productId=SS1031`.

[75] Sony. `http://www.sony.co.in/local/product/playstation+eye`.

[76] optitrack. optitrack. `http://optitrack.com`.

[77] Itzhack Y Bar-Itzhack and Richard R Harman. Optimized triad algorithm for attitude determination. *Journal of guidance, control, and dynamics*, 20(1):208–211, 1997.

[78] Li Wang, Zheng Zhang, and Ping Sun. Quaternion-based kalman filter for ahrs using an adaptive-step gradient descent algorithm. *International Journal of Advanced Robotic Systems*, 12(9):131, 2015.

[79] Jan Bender, Matthias Müller, Miguel A Otaduy, Matthias Teschner, and Miles Macklin. A survey on position-based simulation methods in computer graphics. In *Computer graphics forum*, volume 33, pages 228–251. Wiley Online Library, 2014.

[80] Miles Macklin and Matthias Müller. Position based fluids. *ACM Transactions on Graphics (TOG)*, 32(4):104, 2013.

[81] Thomas Jakobsen. Advanced character physics. In *Game Developers Conference*, volume 3, 2001.

[82] Underwater cable reel simulation video. `https://youtu.be/DO-x2RaZHso`.

[83] RN Marshall, RK Jensen, and GA Wood. A general newtonian simulation of an n-segment open chain model. *Journal of Biomechanics*, 18(5):359–367, 1985.

[84] Brad Buckham, Frederick R Driscoll, and Meyer Nahon. Development of a finite element cable model for use in low-tension dynamics simulation. *Journal of Applied Mechanics*, 71(4):476–485, 2004.

[85] Zhibo Wang and Gang Sun. Parameters influence on maneuvered towed cable system dynamics. *Applied Ocean Research*, 49:27–41, 2015.

[86] C Lambert, M Nahon, B Buckham, and M Seto. Dynamics and control of towed underwater vehicle system, part ii: model validation and turn maneuver optimization. *Ocean engineering*, 30(4):471–485, 2003.

[87] Francisco González, Amelia de la Prada, Alberto Luaces, and Manuel González. Real-time simulation of cable pay-out and reel-in with towed fishing gears. *Ocean Engineering*, 131:295–307, 2017.

[88] CM Ablow and S Schechter. Numerical simulation of undersea cable dynamics. *Ocean engineering*, 10(6):443–457, 1983.

[89] JJ Burgess et al. Bending stiffness in a simulation of undersea cable deployment. *International Journal of Offshore and Polar Engineering*, 3(03), 1993.

[90] JI Gobat and MA Grosenbaugh. Time-domain numerical simulation of ocean cable structures. *Ocean Engineering*, 33(10):1373–1400, 2006.

[91] Sairam Prabhakar and Bradley Buckham. Dynamics modeling and control of a variable length remotely operated vehicle tether. In *OCEANS, 2005. Proceedings of MTS/IEEE*, pages 1255–1262. IEEE, 2005.

[92] Cable component in unreal engine 4. `https://docs.unrealengine.com/latest/INT/Engine/Components/Rendering/CableComponent/`.

[93] Loup Verlet. Computer" experiments" on classical fluids. i. thermodynamical properties of lennard-jones molecules. *Physical review*, 159(1):98, 1967.

[94] Openrov. `https://www.openrov.com/`.

[95] Cable simulation video. `https://youtu.be/_QoMUSlQCsg`.

[96] Jupyter. `http://jupyter.org/`.

[97] Carol Lalli and Timothy R Parsons. *Biological oceanography: an introduction*. Butterworth-Heinemann, 1997.

[98] AS Brierley. *CURRENT BIOLOGY*, volume 27, pages R478 – R483. CELL PRESS, 11 edition, 6 2017.

[99] J Werdell, CS Roesler, and JI Goes. Evaluating long-term changes in phytoplankton community composition using an ocean reflectance inversion model: A case study in the northern arabian sea to explore the emerging frontier of hyperspectral ocean color. In *American Geophysical Union, Ocean Sciences Meeting 2016, abstract# IS12A-03*, 2016.

[100] Elisa Capuzzo, David Stephens, Tiago Silva, Jon Barry, and Rodney M Forster. Decrease in water clarity of the southern and central north sea during the 20th century. *Global change biology*, 21(6):2206–2214, 2015.

[101] Vladimir I Haltrin. Chlorophyll-based model of seawater optical properties. *Applied Optics*, 38(33):6826–6832, 1999.

[102] Jules S Jaffe, Kad D Moore, John McLean, and Michael P Strand. Underwater optical imaging: status and prospects. *Oceanography*, 14(3):66–76, 2001.

[103] Robin M Pope and Edward S Fry. Absorption spectrum (380–700 nm) of pure water. ii. integrating cavity measurements. *Applied optics*, 36(33):8710–8723, 1997.

[104] Hendrik Buiteveld, JHM Hakvoort, and M Donze. Optical properties of pure water. In *Ocean Optics XII*, volume 2258, pages 174–184. International Society for Optics and Photonics, 1994.

[105] LN Thibos, A Bradley, DL Still, X Zhang, and PA Howarth. Theory and measurement of ocular chromatic aberration. *Vision research*, 30(1):33–49, 1990.

[106] John E Tyler. A survey of experimental hydrologic optics. In *Radiative Energy Transfer*, pages 339–354. Elsevier, 1968.

[107] Unreal engine 4 chromatic aberration. `https://docs.unrealengine.com/latest/INT/Engine/Rendering/PostProcessEffects/SceneFringe/`.

[108] Unreal engine 4 refraction. `https://docs.unrealengine.com/latest/INT/Engine/Rendering/Materials/HowTo/Refraction/`.

[109] Exponential height fog user guide. `https://docs.unrealengine.com/latest/INT/Engine/Actors/FogEffects/HeightFog/index.html`.

[110] Natgeopauly-freshwater fish feeding on plankton "peacock bass". `https://www.youtube.com/watch?v=JAd2jZuNkd8`.

[111] Unreal engine 4 particle systems. `https://docs.unrealengine.com/latest/INT/Engine/Rendering/ParticleSystems/`.

[112] Will J Schroeder, Bill Lorensen, and Ken Martin. *The visualization toolkit: an object-oriented approach to 3D graphics*. Kitware, 2004.

[113] Paul Fahlstrom and Thomas Gleason. *Introduction to UAV systems*. John Wiley & Sons, 2012.

[114] Underwater simulation. `https://github.com/uji-ros-pkg/underwater_simulation`.

[115] Olivier Kermorgant. A dynamic simulator for underwater vehicle-manipulators. In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, pages 25–36. Springer, 2014.

[116] Multi-rov training simulator. `http://marinesimulation.com/rovsim-gen3/`.

[117] Myo Myint, Kenta Yonemori, Khin Nwe Lwin, Akira Yanou, and Mamoru Minami. Dual-eyes vision-based docking system for autonomous underwater vehicle: an approach and experiments. *Journal of Intelligent & Robotic Systems*, 92(1):159–186, 2018.

[118] Myo Myint, Kenta Yonemori, Akira Yanou, Shintaro Ishiyama, and Mamoru Minami. Robustness of visual-servo against air bubble disturbance of underwater vehicle system using three-dimensional marker and dual-eye cameras. In *OCEANS'15 MTS/IEEE Washington*, pages 1–8. IEEE, 2015.

[119] Sympy Classical Mechanics. Kane method implementation. http://docs.sympy.org/latest/modules/physics/mechanics/kane.html.

[120] Sympy. http://docs.sympy.org.

[121] Thomas R Kane and David A Levinson. *Dynamics, theory and applications*, chapter 7, pages 30–31. McGraw Hill, 1985.

[122] Thomas R Kane and David A Levinson. *Dynamics, theory and applications*. McGraw Hill, 1985.

[123] George Keith Batchelor. *An introduction to fluid dynamics*. Cambridge university press, 2000.

[124] Thomas R Kane and David A Levinson. *Dynamics, theory and applications*, chapter 6, pages 158–159. McGraw Hill, 1985.

[125] Sympy mechanics. http://docs.sympy.org/latest/modules/physics/mechanics/index.html.

[126] Thomas R Kane and David A Levinson. *Dynamics, theory and applications*, chapter 7, pages 204–205. McGraw Hill, 1985.

[127] Ocean floor environment. https://www.unrealengine.com/marketplace/ocean-floor-environment.

[128] Unreal engine 4 simplegrasswind. https://docs.unrealengine.com/en-us/Engine/Rendering/Materials/Functions/Reference/WorldPositionOffset.

[129] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. Orb: An efficient alternative to sift or surf. In *Computer Vision (ICCV), 2011 IEEE international conference on*, pages 2564–2571. IEEE, 2011.

[130] Heiko Hirschmuller. Stereo processing by semiglobal matching and mutual information. *IEEE Transactions on pattern analysis and machine intelligence*, 30(2):328–341, 2008.

[131] Jianbo Shi and Carlo Tomasi. Good features to track. Technical report, Cornell University, 1993.

[132] Stan Birchfield and Carlo Tomasi. A pixel dissimilarity measure that is insensitive to image sampling. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(4):401–406, 1998.

[133] Opencv stereosgbm. https://docs.opencv.org/trunk/d2/d85/classcv_1_1StereoSGBM.html.

[134] Melike Erol-Kantarci, Hussein T Mouftah, and Sema Oktug. A survey of architectures and localization techniques for underwater acoustic sensor networks. *IEEE Communications Surveys & Tutorials*, 13(3):487–502, 2011.

[135] Peter Corke, Carrick Detweiler, Matthew Dunbabin, Michael Hamilton, Daniela Rus, and Iuliu Vasilescu. Experiments with underwater robot localization and tracking. In *Robotics and Automation, 2007 IEEE International Conference on*, pages 4556–4561. IEEE, 2007.

[136] Asynchronous localization with mobility prediction for underwater acoustic sensor networks. *IEEE Transactions on Vehicular Technology*, 67(3):2543–2556, 2018.

[137] Liam Paull, Sajad Saeedi, Mae Seto, and Howard Li. Auv navigation and localization: A review. *IEEE Journal of Oceanic Engineering*, 39(1):131–149, 2014.

[138] Christian Mai, Simon Pedersen, Leif Hansen, Kasper L Jepsen, and Zhenyu Yang. Subsea infrastructure inspection: A review study. In *Underwater System Technology: Theory and Applications (USYS), IEEE International Conference on*, pages 71–76. IEEE, 2016.

[139] Josep Aulinas, Marc Carreras, Xavier Llado, Joaquim Salvi, Rafael Garcia, Ricard Prados, and Yvan R Petillot. Feature extraction for underwater visual slam. In *OCEANS, 2011 IEEE-Spain*, pages 1–7. IEEE, 2011.

[140] Dominik Honegger, Lorenz Meier, Petri Tanskanen, and Marc Pollefeys. An open source and open hardware embedded metric optical flow cmos camera for indoor and outdoor applications. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 1736–1741. IEEE, 2013.

[141] Jae-Chern Yoo and Tae Hee Han. Fast normalized cross-correlation. *Circuits, systems and signal processing*, 28(6):819, 2009.

[142] Robert Penoyer. The alpha-beta filter. *C Users Journal*, 11(7):73–86, 1993.

[143] Steven W Moore, Harry Bohm, and Vickie Jensen. *Underwater robotics: science, design & fabrication*. Marine Advanced Technology Education (MATE) Center, 2010.

[144] Ms5837-30ba pressure sensor. `https://www.te.com/commerce/DocumentDelivery/DDEController?Action=showdoc&DocId=Data+Sheet%7FMS5837-30BA%7FB1%7Fpdf%7FEnglish%7FENG_DS_MS5837-30BA_B1.pdf%7FCAT-BLPS0017`.

[145] Bluerov2. `https://www.bluerobotics.com/store/rov/bluerov2/bluerov2/`.

[146] Rov mechanical architecture. `https://github.com/uc-csse/RovVision/wiki/Mechanical-Connections-and-Setup`.

[147] Rov software architecture. `https://github.com/uc-csse/RovVision/wiki/Software-Arch`.

[148] Upsquared. `https://up-board.org/upsquared/specifications/`.

[149] `http://www.teanamarina.co.nz/`.

[150] Simulation video. `https://www.csse.canterbury.ac.nz/dronesimlab/videos/simulated_experiment.webm`.

[151] Experiment video. `videohttps://www.csse.canterbury.ac.nz/dronesimlab/videos/localization_seaweed_181105.webm`.

[152] Underwater simulation. `https://youtu.be/MP2xG_Tms3E`, 2018.

[153] Physical water surface. `https://www.unrealengine.com/marketplace/en-US/slug/physical-water-surface`.

[154] Javier Sánchez and Jean-Michel Morel. Motion smoothing strategies for 2d video stabilization. *SIAM Journal on Imaging Sciences*, 11(1):219–251, 2018.

[155] Sea scenario of video stabilization simulation. https://www.youtube.com/watch?v=klSw-7V1_M4, 2018.

[156] Prerna Mishra, Urmila Shrawankar, VS Bivde, P Sarasu, Martin Magdin, Milan Turcani, Lukas Hudec, and Oscar Sanjuan-Martinez. Comparison between famous game engines and eminent games. *International Journal of Interactive Multimedia and Artificial Inteligence*, 4(Special Issue on Artificial Intelligence Underpinning), 2016.

[157] *Unreal Engine 4*. http://www.unrealengine.com.

[158] *Blender Game Engine*. http://www.blender.org/manual/game_engine/introduction.html.

[159] *CryEngine*. https://www.cryengine.com/.

[160] *Panda3d*. http://www.panda3d.org.

[161] *unity*. http://unity3d.com.

[162] *Lumberyard*. https://aws.amazon.com/lumberyard/.

Part V

Appendixes

Appendix A

ROV dynamics implementation

In this Appendix we will present a step by step detailed explanation for the simulation of the ROV dynamics. The framework we used for developing the simulation was the Jupyter notebook, an interactive data science and scientific computing platform [96] mainly for the python language. The core tool for this rigid body dynamic simulation is the SymPy Mechanics, a mechanics python package which generates the necessary symbolic equations of motions for our rigid body system [125]. Some additional supporting packages where also used mainly for plotting and for function generation. The full notebook can be accessed and downloaded for further referencing [48].

The following setup shows the necessary software package tools used in this simulation. The abbreviation "me" is used as a synonym for the SymPy mechanics package for compactness.

Listing A.1: Setup

```
%matplotlib nbagg
import numpy as np
import matplotlib.pyplot as plt
from sympy import *
import sympy.physics.mechanics as me
from sympy import sin, cos, symbols, solve
from pydy.codegen.ode_function_generators import generate_ode_function
from scipy.integrate import odeint
from IPython.display import SVG
me.init_vprinting(use_latex='mathjax')
```

Next we define the inertial reference frame, the generalized coordinates which are the ROV position and orientation and the subsequent generalized speeds which are the angular and linear velocities.

Listing A.2: Symbols definitions

```
N = me.ReferenceFrame('N') # Inertial Reference Frame
O = me.Point('O') # Define a world coordisympynate origin
O.set_vel(N, 0) # Setting world velocity to 0

#generelized coordinates
#q0..3 = xyz positions  q4..6 = yaw,pitch,roll rotations
q = list(me.dynamicsymbols('q0:6'))

#generlized speeds
u = list(me.dynamicsymbols('u0:6'))

kin_diff=Matrix(q).diff()-Matrix(u)
```

In addition we need to define the constants symbols used in our simulation where the geometric aspect of the symbols can be seen in figure 7.2:

Listing A.3: Constant Symbols Definition

```
Wx = symbols('W_x')
Wh = symbols('W_h')
T1 = symbols('T_1')
T2 = symbols('T_2')
Bh = symbols('B_h')
Bw = symbols('B_w')
m_b = symbols('m_b') # Mass of the body
v_b = symbols('v_b') # Volume of the body
mu = symbols('\mu') #drag
mu_r = symbols('\mu_r') #rotational drag
g = symbols('g')
I = list(symbols('Ixx, Iyy, Izz')) #Moments of inertia of body
```

The ROV reference frame is defined by subsequent Euler angles rotations with respect to the inertial frame:

Listing A.4: ROV reference frame

```
Rz=N.orientnew('R_z', 'Axis', (q[3+2], N.z))
Rz.set_ang_vel(N,u[3+2]*N.z)

Ry=Rz.orientnew('R_y', 'Axis', (q[3+1], Rz.y))
Ry.set_ang_vel(Rz,u[3+1]*Rz.y)

R=Ry.orientnew('R', 'Axis', (q[3+0], Ry.x))
R.set_ang_vel(Ry,u[3+0]*Ry.x)
```

In order to apply gravity and buoyancy forces we need to define the COM (Center Of Mass) and COB (Center Of Buoyancy) in the reference frame.

Listing A.5: COM and COB in the reference frame R

```
# Center of mass of body
COM = O.locatenew('COM', q[0]*N.x + q[1]*N.y + q[2]*N.z)

# Set the velocity of COM
COM.set_vel(N, u[0]*N.x + u[1]*N.y + u[2]*N.z)

# center of bouyency
COB = COM.locatenew('COB', R.x*Bw+R.z*Bh)
COB.v2pt_theory(COM, N, R);
```

As an input to the Kane method [119] we need to calculate the inertia dyadic or matrix. The next listing defines the rigid body for the Kane equations with the relevant inertia symbols:

Listing A.6: Rigid Body defenitions

```
Ib = me.inertia(R, *I , ixy=0, iyz=0, izx=0)
Body = me.RigidBody('Body', COM, R, m_b, (Ib, COM))
```

Next we define the drag forces that influencing the underwater ROV

Listing A.7: Rigid Body Drag

```
v=N.x*u[0]+N.y*u[1]+N.z*u[2]
Fd=-v*mu
T_z=(R,-u[3+2]*N.z*mu_r)
T_x=(R,-u[3+0]*Rz.x*mu_r)
T_y=(R,-u[3+1]*Rx.y*mu_r) #rotaional dumping Torqe
```

The following are the buoyancy and gravity constant forces and the thrusters forces. The definition of theses forces is strait forward as can be seen in the following listing:

Listing A.8: External Forces

```
Fg = -N.z * m_b * g # gravity
Fb = N.z * v_b * 1e3 *g # buoyancy
F1, F2, F3 = symbols('f_1, f_2, f_3') # thrusters
```

The next listing shows the final kane's step implemented in the simulation including the integration step:

Listing A.9: Kanes Final Step

```
#multiplying by inverse mass matrix:
u_dot=kane.mass_matrix.inv()*kane.forcing

#replacing the constants with actual numerical values:
subs=[(Wx,0.1), (Wh,0.15), (T1,0.1), (T2,0.05),
    (Bh,0.08), (Bw,0.01), (m_b,1.0), (v_b,0.001),
    (mu,0.3), (mu_r,0.2), (g,9.8), (I[0],0.5),
    (I[1],0.5), (I[2],0.5) ]
u_dot_simp=u_dot.subs(subs)
u_dot_simp=trigsimp(u_dot_simp)

#generating a lambda function to compute the values of the u_dot vector
from sympy import lambdify
def get_next_state_lambda(subs):
    u_dot_simp_q_u_f=u_dot_simp.subs(subs)
    return lambdify((q,u,F1,F2,F3),u_dot_simp_q_u_f)

#integrating function which takes the current state and returns the next
   state
def get_next_state(curr_q,curr_u,control,curr_t,dt,lamb):
    forces=control(curr_t)
    u_dot_f=lamb(curr_q,curr_u,*forces).flatten()
    next_q=curr_q+curr_u*dt
    next_u=curr_u+u_dot_f*dt
    return next_q,next_u
```

We can see the the kane's methods provides us with the final $\dot{u}$ term which we can be use as seen by the previous listing. The next equations summarize the final result of our dynamic simulation. This equation obtained automatically from the SymPy framework after replacing the the constant symbols with the matching numerical values:

$$\dot{u}(u,q,f) = \begin{bmatrix} 1.0f_1\cos(q_4)\cos(q_5) + 1.0f_2\cos(q_4)\cos(q_5) + 1.0f_3(\sin(q_3)\sin(q_5) + \sin(q_4)\cos(q_3)\cos(q_5)) - 0.3u_0 \\ 1.0f_1\sin(q_5)\cos(q_4) + 1.0f_2\sin(q_5)\cos(q_4) + 1.0f_3(-\sin(q_3)\cos(q_5) + \sin(q_4)\sin(q_5)\cos(q_3)) - 0.3u_1 \\ -1.0f_1\sin(q_4) - 1.0f_2\sin(q_4) + 1.0f_3\cos(q_3)\cos(q_4) - 0.3u_2 \\ \frac{2.0\left(-\frac{0.2u_3}{\cos(q_4)} + 0.5u_4u_5 - 0.784\sin(q_3)\right)}{\cos(q_4)} + \frac{2.0(-0.1f_1\cos(q_3)\cos(q_4) + 0.1f_2\cos(q_3)\cos(q_4) + 0.05f_3\sin(q_3)\cos(q_4) + 0.5u_3u_4\cos(q_4) + 0.2u_3\sin(q_4) - 0.2u_5)\sin(q_4)}{\cos^2(q_4)} \\ 0.2f_1\sin(q_3) - 0.2f_2\sin(q_3) + 0.1f_3\cos(q_3) - 1.0u_3u_5\cos(q_4) - 0.4u_4 - 1.568\sin(q_4)\cos(q_3) - 0.196\cos(q_4) \\ -\frac{0.2f_1\cos(q_3)}{\cos(q_4)} + \frac{0.2f_2\cos(q_3)}{\cos(q_4)} + \frac{0.1f_3\sin(q_3)}{\cos(q_4)} + \frac{1.0u_3u_4}{\cos(q_4)} + 1.0u_4u_5\tan(q_4) - \frac{0.4u_5}{\cos^2(q_4)} - 1.568\sin(q_3)\tan(q_4) \end{bmatrix} \quad \text{(A.1)}$$

We can see from this equation that $\dot{u}$ is a function of u,q and f and can be used iteratively as the integration step. finally, in order to use the dynamic simulation inside a full scale 3d simulation we only need to save the mid products of the simulation to a packed file. In the python programming language this can be done using the pickle library. The file can be late uploaded to memory and we can regenerate the lambda function for getting the $\dot{u}$ vector for the iterative integration.

Appendix B

Game Engines General Background and comparison

There are several game engines in the market and also a lot of comparisons in the form of papers [156], videos showcases, etc. For the purpose of this research, we chose the UE4. This decision was made at the beginning of the research and as of now (2019) it is still one of the best realistic game engines on the market. Below are some of the game engines we looked at.

1. UE4 [157] - **Pros:** it can create incredibly realistic environments especially when using the open world package, actively developed, large community, large market place, free license for academic and scientific purposes. **Cons:** lack of decent scripting programming language they are using their propriety visual node-based language which they call blueprints.

2. BGE Blender game engine [158] - **Pros:** It is based on Blender and offers very useful scripting capabilities in Python. It is an open source project and actively developed. **Cons:** The primary focus of blender seems to be on creating static reach content and post-processing for creating a live scene, but not real-time gaming. games created in Blender Game engine does not look realistic enough.

3. CryEngine [159] - **Pros:** One of the most realistic game engines. It has very realistic visuals and interfaces with the Lua scripting language. The engine is actively developed and has a large community. free license for academic and scientific purposes (In the form of "Serious Game") **Cons:** Recently Crytek (The company behind CryEngine) is scaling done which raises some concerns about the future of this Game Engine.

4. Panda3d [160] - **Pros:** Fully open source python based game engine. Provides fully programmable and controllable interface. **Cons:** small community compared to other game engines and it seems that it not actively developed.

5. Unity Game engine [161] - **Pros:** very powerful game engine with C# interfaces and full control over the rendering pipeline. It is very popular with emphasis on multi-platform capabilities and a large marketplace. **Cons:** Although there is a Linux version it is not very stable.

6. Lumberyard [162] - This is a new game engine by Amazon. It is based on CryEngine so It inherits a lot of CryEngine capabilities. **Pros:** A large company behind the product usually grantees a lot of support and documentation resulting in a more stable product. **Cons:** relatively new and lack of proper support in platforms like Linux. Amazon is currently putting a lot of effort into this engine so I will keep an eye on its development.

7. Gazebo [19] - actually not a game engine rather a robot simulation environment. **Pros:** very convenient to use with scientific orientation and already used for simulating quadrotors UAVs **Cons:** It is not meeting the current game engine standards of realism.

8. FlightGear - an open-source flight simulator with a wide variety of API's and flight dynamic models. **Pros:** It is suitable for rendering large terrain from a distance. **Cons:** More focused on flight dynamics then realistic rendering of 3D scenes. [52]